Министерство науки и высшего образования РФ
ФГБОУ ВО «Ульяновский государственный университет»
Факультет математики, информационных и авиационных технологий

Перцев А.А.

«ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ. МЕТОДЫ И АЛГОРИТМЫ»

Методические указания для самостоятельной работы студентов

для студентов бакалавриата по направлениям
09.03.03 Прикладная информатика, 02.03.03 Математическое обеспечение и администрирование информационных систем и студентов, обучающихся по программе магистратуры 02.04.03 Математическое обеспечение и администрирование информационных систем

Ульяновск, 2019

Методические указания для самостоятельной работы по дисциплине «Параллельное программирование» для студентов старших курсов бакалавриата по направлениям: 09.03.03 Прикладная информатика, 02.03.03 Математическое обеспечение и администрирование информационных систем и студентов, обучающихся по программе магистратуры 02.04.03 Математическое обеспечение и администрирование информационных систем/ составитель: Перцев А.А. – Ульяновск: УлГУ, 2019.

Настоящие методические указания предназначены для студентов старших бакалавриата по направлениям 09.03.03 Прикладная информатика, 02.03.03 Математическое обеспечение и администрирование информационных систем и студентов, обучающихся по программе магистратуры 02.04.03 Математическое обеспечение и администрирование информационных систем.

В работе приведены литература по дисциплине, примеры кода для реализации лабораторных работ, методические указания для самостоятельной работы студентов и задания для самостоятельного выполнения.

Методические указания будут полезны при подготовке к лабораторным занятиям и к экзамену/зачету.

Рекомендованы к введению в образовательный процесс Ученым советом Факультета математики, информационных и авиационных технологий УлГУ (протокол № 2/19 от 19 марта 2019 г.).

1. ЛИТЕРАТУРА ДЛЯ ИЗУЧЕНИЯ ДИСЦИПЛИНЫ

1. Гергель В.П. Теория и практика параллельных вычислений : учеб. пособие. - Москва : Интернет-Ун-т Информ. Технологий : БИНОМ : Лаборатория знаний, 2007. - 423 с. - (Основы информационных технологий). - Библиогр.: с. 418-423. - ISBN 978-5-9556-0096-3 (ИНТУИТ.РУ) (в пер.). - ISBN 978-5-94774-645-7 (БИНОМ.ЛЗ) (в пер.)
2. Зыков, С. В. Программирование. Объектно-ориентированный подход : учебник -5-534-00850-0. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://urait и практикум для академического бакалавриата / С. В. Зыков. — Москва : Издательство Юрайт, 2019. — 155 с. — (Бакалавр. Академический курс). — ISBN 978.ru/bcode/434106
3. Зыков, С. В. Программирование. Функциональный подход : учебник и практикум для академического бакалавриата / С. В. Зыков. — Москва : Издательство Юрайт, 2019. — 164 с. — (Бакалавр. Академический курс). — ISBN 978-5-534-00844-9. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://urait.ru/bcode/434613
4. Жаркова Г.А. Методы программирования и прикладные алгоритмы : учеб.-метод. пособие / Жаркова Г.А., А. В. Жарков; УлГУ, ФМИиАТ. - Ульяновск: УлГУ, 2018. - 96 с.
5. Малявко, А. А. Параллельное программирование на основе технологий OpenMP, MPI, CUDA : учебное пособие для академического бакалавриата / А. А. Малявко. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2019. — 129 с. — (Высшее образование). — ISBN 978-5-534-11827-8. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://urait.ru/bcode/446247
6. Параллельные вычисления общего назначения на графических процессорах: учебное пособие / К.А. Некрасов [и др.].. — Екатеринбург : Уральский федеральный университет, ЭБС АСВ, 2016. — 104 c. — ISBN 978-5-7996-1722-6. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/69657
7. Туральчук К.А. Параллельное программирование с помощью языка C# / Туральчук К.А.. — Москва : Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Эр Медиа, 2019. — 189 c. — ISBN 978-5-4486-0506-2. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/79714

2. МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Ниже приведены методические указания для самостоятельной подготовки лабораторных работ.

СПИСОК ЛАБОРАТОРНЫХ РАБОТ

Лабораторная работа №1: «Параллельные алгоритмы матрично-векторного умножения» 5

Лабораторная работа №2: «Параллельные алгоритмы матричного умножения» 15

Лабораторная работа №3: «Параллельные методы решения систем линейных уравнений» 3

Лабораторная работа №4: «Параллельные методы сортировки данных» 20

Лабораторная работа №5: «Параллельные алгоритмы обработки графов» 32

Лабораторная работа №6: «Параллельные алгоритмы решения дифференциальных уравнений в частных производных» 44

Лабораторная работа №1: «Параллельные алгоритмы матрично-векторного умножения»

Матрицы и матричные операции широко используются при математическом моделировании самых разнообразных процессов, явлений и систем. Матричные вычисления составляют основу многих научных и инженерных расчетов – среди областей приложений могут быть указаны вычислительная математика, физика, экономика и др.

Являясь вычислительно-трудоемкими, матричные вычисления представляют собой классическую область применения параллельных вычислений. С одной стороны, использование высокопроизводительных многопроцессорных систем позволяет существенно повысить сложность решаемых задач. С другой стороны, в силу своей достаточно простой формулировки матричные операции предоставляют прекрасную возможность для демонстрации многих приемов и методов параллельного программирования.

Целью данной лабораторной работы является разработка параллельной программы, которая выполняет умножение матрицы на вектор. Выполнение лабораторной включает выполнение следующих задач:

1. Постановка задачи матрично-векторного умножения
2. Реализация последовательного алгоритма умножения матрицы на вектор
3. Разработка параллельного алгоритма умножения матрицы на вектор
4. Реализация параллельного алгоритма матрично-векторного умножения

Псевдокод для представленного алгоритма умножения матрицы на вектор может выглядеть следующим образом:

```
// Serial algorithm of matrix-vector multiplication
for (i = 0; i < m; i++){

  c[i] = 0;

  for (j = 0; j < n; j++){
    c[i] += A[i][j]*b[j]

  }

}
```

При выполнении **задачи 2** необходимо реализовать последовательный алгоритм матрично-векторного умножения. Начальный вариант будущей программы представлен. Он содержит часть исходного кода, в котором заданы необходимые параметры проекта. В ходе выполнения задания необходимо дополнить имеющийся вариант программы операциями ввода размера объектов, инициализации матрицы и вектора, умножения матрицы на вектор и вывода результатов.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>

// Function for simple definition of matrix and vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables
```

```
  for (i=0; i<Size; i++) {
    pVector[i] = 1;
    for (j=0; j<Size; j++)
      pMatrix[i*Size+j] = i;
  }
}

// Function for random definition of matrix and vector elements
void RandomDataInitialization(double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i=0; i<Size; i++) {
    pVector[i] = rand()/double(1000);
    for (j=0; j<Size; j++)
      pMatrix[i*Size+j] = rand()/double(1000);
  }
}

// Function for memory allocation and definition of object's elements
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int &Size) {
  // Size of initial matrix and vector definition
  do {
    printf("\nEnter size of the initial objects: ");
    scanf("%d", &Size);
    printf("\nChosen objects size = %d\n", Size);
      if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  }
  while (Size <= 0);
  // Memory allocation
  pMatrix = new double [Size*Size];
  pVector = new double [Size];
  pResult = new double [Size];
  // Definition of matrix and vector elements
  DummyDataInitialization(pMatrix, pVector, Size);
}

// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
  int i, j; // Loop variables
  for (i=0; i<RowCount; i++) {
    for (j=0; j<ColCount; j++)
        printf("%7.4f ", pMatrix[i*RowCount+j]);

      printf("\n");
  }
}

// Function for formatted vector output
void PrintVector (double* pVector, int Size) {
  int i;
  for (i=0; i<Size; i++)
    printf("%7.4f ", pVector[i]);
}

// Function for matrix-vector multiplication
void ResultCalculation(double* pMatrix, double* pVector, double* pResult,
  int Size) {
  int i, j;  // Loop variables
  for (i=0; i<Size; i++) {
    pResult[i] = 0;
```

```
      for (j=0; j<Size; j++)
        pResult[i] += pMatrix[i*Size+j]*pVector[j];
  }
}

// Function for computational process termination
void ProcessTermination(double* pMatrix,double* pVector,double* pResult) {
  delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
}


void main() {
  double* pMatrix; // The first argument - initial matrix
  double* pVector;  // The second argument - initial vector
  double* pResult;  // Result vector for matrix-vector multiplication
  int Size;           // Sizes of initial matrix and vector
  time_t start, finish;
  double duration;

  printf("Serial matrix-vector multiplication program\n");
  // Memory allocation and definition of objects' elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // Matrix and vector output
  printf ("Initial Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector \n");
  PrintVector(pVector, Size);

  // Matrix-vector multiplication
  start = clock();
  ResultCalculation(pMatrix, pVector, pResult, Size);
  finish = clock();
  duration = (finish-start)/double(CLOCKS_PER_SEC);

  // Printing the result vector
  printf ("\n Result Vector: \n");
  PrintVector(pResult, Size);

  // Printing the time spent by matrix-vector multiplication
  printf("\n Time of execution: %f\n", duration);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
```

Принципы распараллеливания

Разработка алгоритмов (а в особенности методов параллельных вычислений) для решения сложных научно-технических задач часто представляет собой значительную проблему. Здесь же мы будем полагать, что вычислительная схема решения нашей задачи умножения матрицы на вектор уже известна. Действия для определения эффективных способов организации параллельных вычислений могут состоять в следующем:

- Выполнить анализ имеющейся вычислительной схемы и осуществить ее разделение (*декомпозицию*) на части (*подзадачи*), которые могут быть реализованы в значительной степени независимо друг от друга,

- Выделить для сформированного набора подзадач *информационные взаимодействия*, которые должны осуществляться в ходе решения исходной поставленной задачи,
- Определить необходимую (или доступную) для решения задачи *вычислительную систему* и выполнить *распределение* имеющегося набора подзадач между процессорами системы.

Такие этапы разработки параллельных алгоритмов впервые были предложены Фостером (I. Foster)

При самом общем рассмотрении понятно, что объем вычислений для каждого используемого процессора должен быть примерно одинаков – это позволит обеспечить равномерную вычислительную загрузку (*балансировку*) процессоров. Кроме того, также понятно, что распределение подзадач между процессорами должно быть выполнено таким образом, чтобы наличие информационных связей (*коммуникационных взаимодействий*) между подзадачами было минимальным.

Определение подзадач

Для многих методов матричных вычислений характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Данный момент свидетельствует о наличии *параллелизма по данным* при выполнении матричных расчетов и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между процессорами используемой вычислительной системы. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем распределения данных порождает целый ряд *параллельных алгоритмов матричных вычислений*.

Дадим кратко общую характеристику распределения данных для матричных алгоритмов – более подробно данный материал содержится в разделе 7 учебного курса. Наиболее общие и широко используемые способы разделения матриц состоят в разбиении данных на *полосы* (по вертикали или горизонтали) или на прямоугольные фрагменты (*блоки*).

1. **Ленточное разбиение матрицы.** При *ленточном* (*block-striped*) разбиении каждому процессору выделяется то или иное подмножество строк (*rowwise* или *горизонтальное разбиение*) или столбцов (*columnwise* или *вертикальное разбиение*) матрицы Разделение строк и столбцов на полосы в большинстве случаев происходит на *непрерывной* (*последовательной*) основе.

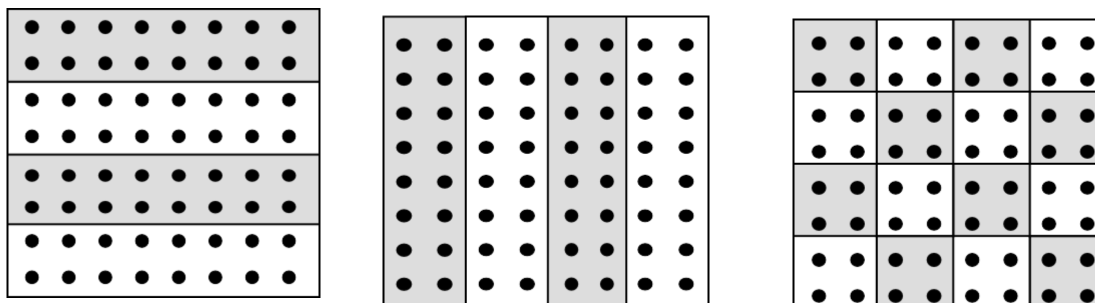
Другой возможный подход к формированию полос состоит в применении той или иной схемы *чередования* (*цикличности*) строк или столбцов. Как правило, для чередования используется число процессоров p .

Циклическая схема формирования полос может оказаться полезной для лучшей балансировки вычислительной нагрузки процессоров (например, при решении системы линейных уравнений с использованием метода Гаусса).

2. **Блочное разбиение матрицы.** При *блочном* (*checkerboard block*) разделении матрица делится на прямоугольные наборы элементов – при этом, как правило, используется разделение на непрерывной основе. Пусть количество процессоров составляет $p = s \cdot q$, количество строк матрицы является кратным s, а количество столбцов – кратным q, то есть

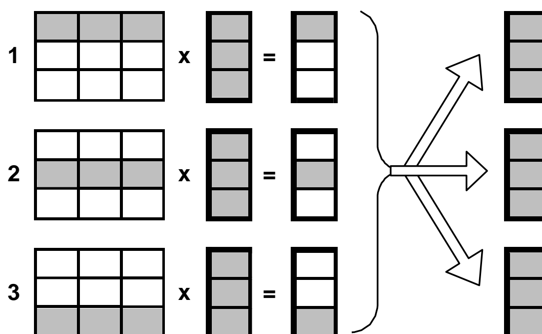
$m = k \cdot s$ и $n = l \cdot q$.

При таком подходе целесообразно, чтобы вычислительная система имела физическую или, по крайней мере, логическую топологию процессорной решетки из *s* строк и *q* столбцов. В этом случае при разделении данных на непрерывной основе процессоры, соседние в структуре решетки, обрабатывают смежные блоки исходной матрицы. Следует отметить, однако, что и для блочной схемы может быть применено циклическое чередование строк и столбцов.



Выделение информационных зависимостей.

В общем виде схема информационного взаимодействия подзадач в ходе выполняемых вычислений показана на рисунке.



Для выполнения базовой подзадачи скалярного произведения процессор должен содержать соответствующую строку матрицы *pMatrix* и копию вектора *pVector*. После завершения вычислений каждая базовая подзадача определяет один из элементов вектора результата *pResult*.

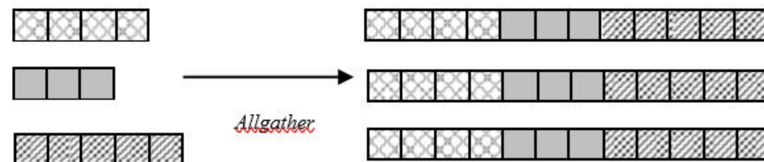
Для объединения результатов расчета и получения полного вектора *pResult* на каждом из процессоров вычислительной системы необходимо выполнить операцию обобщенного сбора данных, в которой каждый процессор передает свой вычисленный элемент вектора *c* всем остальным процессорам.

Масштабирование и распределение подзадач по процессорам

В процессе умножения плотной матрицы на вектор количество вычислительных

операций для получения скалярного произведения одинаково для всех базовых подзадач. Поэтому в случае, когда число процессоров *p* меньше числа базовых подзадач *m* (*p*<*m*), мы можем объединить базовые подзадачи таким образом, чтобы каждый процессор выполнял несколько таких задач, соответствующих непрерывной последовательности строк матрицы *pMatrix*. В этом случае по окончании вычислений каждая базовая подзадача определяет набор элементов результирующего вектора *pResult*.

Распределение подзадач между процессорами вычислительной системы может быть выполнено произвольным образом.



При выполнении **задачи 4** необходимо разработать параллельный алгоритм умножения матрицы на вектор.

Понятие параллельной программы

Под *параллельной программой* в рамках MPI понимается множество одновременно выполняемых *процессов*. Процессы могут выполняться на разных процессорах, но на одном процессоре могут располагаться и несколько процессов (в этом случае их исполнение осуществляется в режиме разделения времени). В предельном случае для выполнения параллельной программы может использоваться один процессор – как правило, такой способ применяется для начальной проверки правильности параллельной программы.

Количество процессов и число используемых процессоров определяется в момент запуска параллельной программы средствами среды исполнения MPI-программ и в ходе вычислений меняться не может (в стандарте MPI-2 предусматривается возможность динамического изменения количества процессов). Все процессы программы последовательно перенумерованы от 0 до *p-1* , где *p* есть общее количество процессов. Номер процесса именуется *рангом* процесса.

Программный код параллельного приложения для умножения матрицы на вектор

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <mpi.h>

int ProcNum = 0;       // Number of available processes
int ProcRank = 0;      // Rank of current process

// Function for simple definition of matrix and vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables

  for (i=0; i<Size; i++) {
    pVector[i] = 1;
    for (j=0; j<Size; j++)
      pMatrix[i*Size+j] = i;
  }
```

```
}

// Function for random definition of matrix and vector elements
void RandomDataInitialization(double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i=0; i<Size; i++) {
    pVector[i] = rand()/double(1000);
    for (j=0; j<Size; j++)
      pMatrix[i*Size+j] = rand()/double(1000);
  }
}

// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcResult,
  int &Size, int &RowNum) {
  int RestRows; // Number of rows, that haven't been distributed yet
  int i;             // Loop variable

  if (ProcRank == 0) {
    do {
      printf("\nEnter size of the initial objects: ");
      scanf("%d", &Size);
      if (Size < ProcNum) {
        printf("Size of the objects must be greater than
                number of processes! \n ");
      }
    }
    while (Size < ProcNum);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  RestRows = Size;
  for (i=0; i<ProcRank; i++)
    RestRows = RestRows-RestRows/(ProcNum-i);
  RowNum = RestRows/(ProcNum-ProcRank);

  pVector = new double [Size];
  pResult = new double [Size];


  pProcRows = new double [RowNum*Size];
  pProcResult = new double [RowNum];

  if (ProcRank == 0) {
    pMatrix = new double [Size*Size];
    RandomDataInitialization(pMatrix, pVector, Size);
  }
}

// Data distribution among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
                      int Size, int RowNum) {
  int *pSendNum; // the number of elements sent to the process
  int *pSendInd; // the index of the first data element sent to the process
  int RestRows=Size; // Number of rows, that haven't been distributed yet

  MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Alloc memory for temporary objects
  pSendInd = new int [ProcNum];
  pSendNum = new int [ProcNum];
```

```
  // Define the disposition of the matrix rows for current process
  RowNum = (Size/ProcNum);
  pSendNum[0] = RowNum*Size;
  pSendInd[0] = 0;
  for (int i=1; i<ProcNum; i++) {
    RestRows -= RowNum;
    RowNum = RestRows/(ProcNum-i);
    pSendNum[i] = RowNum*Size;
    pSendInd[i] = pSendInd[i-1]+pSendNum[i-1];
  }

  // Scatter the rows
  MPI_Scatterv(pMatrix , pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Free the memory
  delete [] pSendNum;
  delete [] pSendInd;
}

// Function for gathering the result vector
void ResultReplication(double* pProcResult, double* pResult, int Size,
    int RowNum) {
  int i;             // Loop variable
  int *pReceiveNum; // Number of elements, that current process sends
  int *pReceiveInd;  /* Index of the first element from current process
                        in result vector */
  int RestRows=Size; // Number of rows, that haven't been distributed yet

  //Alloc memory for temporary objects
  pReceiveNum = new int [ProcNum];
  pReceiveInd = new int [ProcNum];

  //Define the disposition of the result vector block of current processor
  pReceiveInd[0] = 0;
  pReceiveNum[0] = Size/ProcNum;
  for (i=1; i<ProcNum; i++) {
    RestRows -= pReceiveNum[i-1];
    pReceiveNum[i] = RestRows/(ProcNum-i);
    pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1];
  }

  //Gather the whole result vector on every processor
  MPI_Allgatherv(pProcResult, pReceiveNum[ProcRank], MPI_DOUBLE, pResult,
    pReceiveNum, pReceiveInd, MPI_DOUBLE, MPI_COMM_WORLD);

  //Free the memory
  delete [] pReceiveNum;
  delete [] pReceiveInd;
}

// Function for sequential matrix-vector multiplication
void SerialResultCalculation(double* pMatrix, double* pVector, double*
pResult, int Size) {
  int i, j;  // Loop variables
  for (i=0; i<Size; i++) {
    pResult[i] = 0;
    for (j=0; j<Size; j++)
      pResult[i] += pMatrix[i*Size+j]*pVector[j];
  }
}

// Function for calculating partial matrix-vector multiplication
```

```
void ParallelResultCalculation(double* pProcRows, double* pVector, double*
pProcResult, int Size, int RowNum) {
  int i, j;  // Loop variables
  for (i=0; i<RowNum; i++) {
    pProcResult[i] = 0;
    for (j=0; j<Size; j++)
      pProcResult[i] += pProcRows[i*Size+j]*pVector[j];
  }
}

// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
  int i, j; // Loop variables
  for (i=0; i<RowCount; i++) {
    for (j=0; j<ColCount; j++)
      printf("%7.4f ", pMatrix[i*ColCount+j]);
    printf("\n");
  }
}

// Function for formatted vector output
void PrintVector (double* pVector, int Size) {
  int i;
  for (i=0; i<Size; i++)
    printf("%7.4f ", pVector[i]);
}

void TestDistribution(double* pMatrix, double* pVector, double* pProcRows,
    int Size, int RowNum) {
  if (ProcRank == 0) {
    printf("Initial Matrix: \n");
    PrintMatrix(pMatrix, Size, Size);
    printf("Initial Vector: \n");
    PrintVector(pVector, Size);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  for (int i=0; i<ProcNum; i++) {
    if (ProcRank == i) {
      printf("\nProcRank = %d \n", ProcRank);
      printf(" Matrix Stripe:\n");
      PrintMatrix(pProcRows, RowNum, Size);
      printf(" Vector: \n");

      PrintVector(pVector, Size);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

void TestPartialResults(double* pProcResult, int RowNum) {
  int i;    // Loop variables
  for (i=0; i<ProcNum; i++) {
    if (ProcRank == i) {
      printf("\nProcRank = %d \n Part of result vector: \n", ProcRank);
      PrintVector(pProcResult, RowNum);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

void TestResult(double* pMatrix, double* pVector, double* pResult,
  int Size) {
  // Buffer for storing the result of serial matrix-vector multiplication
```

```
  double* pSerialResult;
  // Flag, that shows wheather the vectors are identical or not
  int equal = 0;
  int i;                 // Loop variable

  if (ProcRank == 0) {
    pSerialResult = new double [Size];
    SerialResultCalculation(pMatrix, pVector, pSerialResult, Size);
    for (i=0; i<Size; i++) {
      if (pResult[i] != pSerialResult[i])
        equal = 1;
    }
    if (equal == 1)
      printf("The results of serial and parallel algorithms "
             "are NOT identical. Check your code.");
    else
      printf("The results of serial and parallel algorithms "
             "are identical.");
  }
}

// Function for computational process termination
void ProcessTermination (double* pMatrix, double* pVector, double* pResult,
    double* pProcRows, double* pProcResult) {
  if (ProcRank == 0)
    delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
  delete [] pProcRows;
  delete [] pProcResult;
}

void main(int argc, char* argv[]) {
  double* pMatrix; // The first argument - initial matrix
  double* pVector;  // The second argument - initial vector
  double* pResult;  // Result vector for matrix-vector multiplication
  int Size;          // Sizes of initial matrix and vector
  double* pProcRows;
  double* pProcResult;
  int RowNum;
  double Start, Finish, Duration;

  MPI_Init(&argc, &argv);



 MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
 MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);


 ProcessInitialization(pMatrix, pVector, pResult, pProcRows, pProcResult,
   Size, RowNum);


 Start = MPI_Wtime();

 DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);
 ParallelResultCalculation(pProcRows, pVector, pProcResult, Size, RowNum);
 ResultReplication(pProcResult, pResult, Size, RowNum);

 Finish = MPI_Wtime();
 Duration = Finish-Start;
```

```
  TestResult(pMatrix, pVector, pResult, Size);if
  (ProcRank == 0) {
    printf("Time of execution = %f\n", Duration);
  }

  ProcessTermination(pMatrix, pVector, pResult, pProcRows, pProcResult);

  MPI_Finalize();
}
```

Лабораторная работа №2: «Параллельные алгоритмы матричного умножения»

Операция умножения матриц является одной из основных задач матричных вычислений. В данной лабораторной работе рассматриваются последовательный алгоритм матричного умножения и параллельный алгоритм Фокса (*the Fox algorithm*), основанный на блочной схеме разделения данных.

Целью данной лабораторной работы является разработка параллельной программы, которая выполняет умножение двух квадратных матриц. Выполнение лабораторной включает выполнение следующих задач:

1. Определение задачи матричного умножения
2. Реализация последовательного алгоритма матричного умножения
3. Разработка параллельного алгоритма матричного умножения
4. Реализация параллельного алгоритма умножения матриц

Псевдокод для алгоритма умножения матрицы на вектор может выглядеть следующим образом (здесь и далее предполагается, что матрицы, участвующие в умножении,квадратные, то есть имеют размерность $Size \times Size$):

```
// Serial algorithm of matrix multiplication
for (i=0; i<Size; i++) {
  for (j=0; j<Size; j++) {
    MatrixC[i][j] = 0;
    for (k=0; k<Size; k++) {
      MatrixC[i][j] = MatrixC[i][j] + MatrixA[i][k]*MatrixB[k][j];
    }
  }
}
```

При выполнении **задачи 2** необходимо реализовать последовательный алгоритм матричного умножения. Начальный вариант будущей программы представлен. Он содержит часть исходного кода и в котором заданы необходимые параметры проекта. В ходе выполнения упражнения необходимо дополнить имеющийся вариант программы операциями ввода размера матриц, задание исходных данных, умножения матриц и вывода результатов.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>

// Function for simple initialization of matrix elements
void DummyDataInitialization (double* pAMatrix,double* pBMatrix,int Size) {
  int i, j;  // Loop variables

  for (i=0; i<Size; i++)
    for (j=0; j<Size; j++) {
      pAMatrix[i*Size+j] = 1;
      pBMatrix[i*Size+j] = 1;
  }
}

// Function for random initialization of matrix elements
void RandomDataInitialization (double* pAMatrix, double* pBMatrix,
  int Size) {
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i=0; i<Size; i++)
    for (j=0; j<Size; j++)  {
      pAMatrix[i*Size+j] = rand()/double(1000);
      pBMatrix[i*Size+j] = rand()/double(1000);
  }
}

// Function for memory allocation and initialization of matrix elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, int &Size) {
  // Setting the size of matricies
  do {


    printf("\nEnter size of matricies: ");
    scanf("%d", &Size);
    printf("\nChosen matricies' size = %d\n", Size);
    if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  }
  while (Size <= 0);

  // Memory allocation
  pAMatrix = new double [Size*Size];
  pBMatrix = new double [Size*Size];
  pCMatrix = new double [Size*Size];

  // Initialization of matrix elements
  DummyDataInitialization(pAMatrix, pBMatrix, Size);
  for (int i=0; i<Size*Size; i++) {
    pCMatrix[i] = 0;
  }
}

// Function for formatted matrix output
```

```
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
  int i, j; // Loop variables
  for (i=0; i<RowCount; i++) {
    for (j=0; j<ColCount; j++)
      printf("%7.4f ", pMatrix[i*RowCount+j]);
      printf("\n");
  }
}

// Function for matrix multiplication
void SerialResultCalculation(double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size) {
  int i, j, k;  // Loop variables
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
      for (k=0; k<Size; k++)
        pCMatrix[i*Size+j] += pAMatrix[i*Size+k]*pBMatrix[k*Size+j];
  }
}

// Function for computational process termination
void ProcessTermination (double* pAMatrix, double* pBMatrix,
  double* pCMatrix) {
  delete [] pAMatrix;
  delete [] pBMatrix;
  delete [] pCMatrix;
}

void main() {
  double* pAMatrix; // The first argument of matrix multiplication
  double* pBMatrix;  // The second argument of matrix multiplication
  double* pCMatrix;  // The result matrix
  int Size;            // Size of matricies
  time_t start, finish;
  double duration;

  printf("Serial matrix multiplication program\n");
  // Memory allocation and initialization of matrix elements
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

  // Matrix output
 printf ("Initial A Matrix \n");
 PrintMatrix(pAMatrix, Size, Size);
 printf("Initial B Matrix \n");
 PrintMatrix(pBMatrix, Size, Size);


 // Matrix multiplication
 start = clock();
 SerialResultCalculation(pAMatrix, pBMatrix, pCMatrix, Size);finish =
 clock();
 duration = (finish-start)/double(CLOCKS_PER_SEC);


 // Printing the result matrix printf
 ("\n Result Matrix: \n");
 PrintMatrix(pCMatrix, Size, Size);


 // Printing the time spent by matrix multiplication
 printf("\n Time of execution: %f\n", duration);
```

При построении параллельных способов выполнения матричного умножения наряду с рассмотрением матриц в виде наборов строк и столбцов широко используется блочное представление матриц.

Выделение информационных зависимостей

Итак, за основу параллельных вычислений для матричного умножения при блочном разделении данных принят подход, при котором базовые подзадачи отвечают за вычисления отдельных блоков матрицы C и при этом в подзадачах на каждой итерации расчетов располагаются только по одному блоку исходных матриц A и B . Для нумерации подзадач будем использовать индексы размещаемых в подзадачах блоков матрицы C, т.е. подзадача (i,j) отвечает за вычисление блока C_{ij} – тем самым, набор подзадач образует квадратную решетку, соответствующую структуре блочного представления матрицы C.

В соответствии с алгоритмом Фокса в ходе вычислений на каждой базовой подзадаче (i,j)

располагается четыре матричных блока:

- блок C_{ij} матрицы C, вычисляемый подзадачей;
- блок A_{ij} матрицы A, размещаемый в подзадаче перед началом вычислений;
- блоки A'_{ij} , B'_{ij} матриц A и B, получаемые подзадачей в ходе

выполнения вычислений.

Выполнение параллельного метода включает:

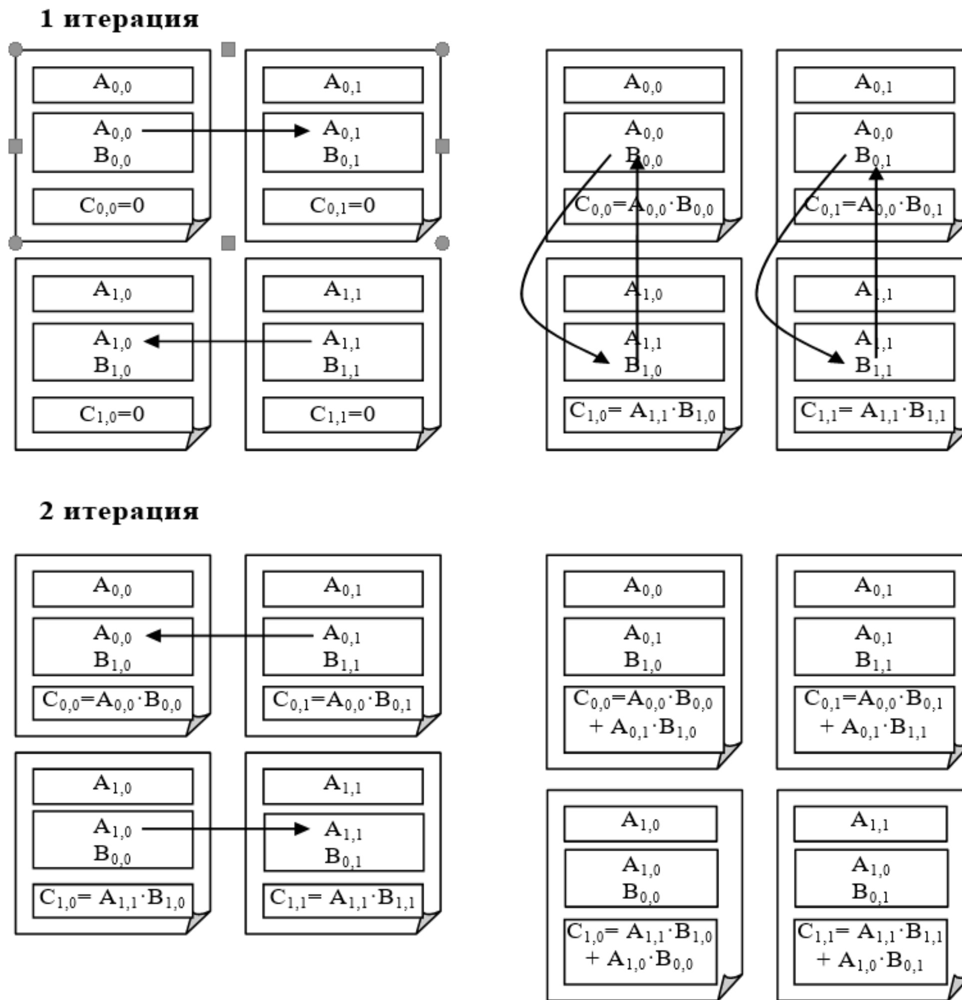
- этап инициализации, на котором каждой подзадаче (i,j) передаются блоки A_{ij}, B_{ij} и обнуляются блоки C_{ij} на всех подзадачах;
- п вычислений, в рамках которого на каждой итерации l, $0\le l<q$, осуществляются следующие операции:
- для каждой строки i, $0\le i<q$, блок A_{ij} подзадачи (i,j) пересылается на все подзадачи той же строки i решетки; индекс j, определяющий положение подзадачи в строке, вычисляется в соответствии с выражением

  $$j = (i+l)\ mod\ q,$$

  где mod есть операция получения остатка от целочисленного деления;
- полученные в результаты пересылок блоки A'_{ij}, B'_{ij} каждой подзадачи (i,j) перемножаются и прибавляются к блоку C_{ij}

  $$C_{ij} = C_{ij} + A'_{ij} \times B'_{ij} ;$$
- и B'_{ij} каждой подзадачи (i,j) пересылаются подзадачам, являющимися соседями сверху в столбцах решетки подзадач (блоки подзадач из первой строки решетки пересылаются подзадачам последней строки решетки).

Для пояснения приведенных правил параллельного метода на рисунке приведено состояние блоков в каждой подзадаче в ходе выполнения итераций этапа вычислений (для решетки подзадач *2×2*).



Масштабирование и распределение подзадач по процессорам

В рассмотренной схеме параллельных вычислений количество блоков может варьироваться в зависимости от выбора размера блоков – эти размеры могут быть подобраны таким образом, чтобы общее количество базовых подзадач совпадало с числом процессоров p. Так, например, в наиболее простом случае, когда число процессоров представимо в виде $p=\delta^2$ (т.е. является полным квадратом) можно выбрать количество блоков в матрицах по вертикали и горизонтали равным δ (т.е. $q=\delta$). Такой способ определения количества блоков приводит к тому, что объем вычислений в каждой подзадаче является одинаковым и, тем самым, достигается полная балансировка вычислительной нагрузки между процессорами. В более общем случае при произвольных количестве процессоров и размеров матриц балансировка вычислений может отличаться от абсолютно одинаковой, но, тем не менее, при надлежащем выборе параметров может быть распределена между процессорами равномерно в рамках требуемой точности.

Для эффективного выполнения алгоритма Фокса, в котором базовые подзадачи представлены в виде квадратной решетки и в ходе вычислений выполняются операции передачи блоков по строкам и столбцам решетки подзадач, наиболее адекватным решением является организация множества имеющихся процессоров также в виде

квадратной решетки. В этом случае можно осуществить непосредственное отображение набора подзадач на множество процессоров – базовую подзадачу (i,j) следует располагать на процессоре Pi,j. Необходимая структура сети передачи данных может быть обеспечена на физическом уровне, если топология вычислительной системы имеет вид решетки или полного графа.

При выполнении **задачи 4** предложено разработать параллельный алгоритм Фокса для матричного умножения.

Программный код параллельного приложения для матричного умножения

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <mpi.h>

int ProcNum = 0;        // Number of available processes
int ProcRank = 0;       // Rank of current process
int GridSize;           // Size of virtual processor grid
int GridCoords[2];      // Coordinates of current processor in grid
MPI_Comm GridComm;      // Grid communicator
MPI_Comm ColComm;       // Column communicator
MPI_Comm RowComm;       // Row communicator

/// Function for simple initialization of matrix elements
void DummyDataInitialization (double* pAMatrix, double* pBMatrix,int Size){
int i, j;  // Loop variables

for (i=0; i<Size; i++)
for (j=0; j<Size; j++) {
pAMatrix[i*Size+j] = 1;
pBMatrix[i*Size+j] = 1;
}
}

// Function for random initialization of matrix elements
void RandomDataInitialization (double* pAMatrix, double* pBMatrix,
int Size) {
int i, j;  // Loop variables
srand(unsigned(clock())); for
(i=0; i<Size; i++)
for (j=0; j<Size; j++) { pAMatrix[i*Size+j] =
rand()/double(1000);pBMatrix[i*Size+j] =
rand()/double(1000);
}
}

// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {

int i, j; // Loop variables for
(i=0; i<RowCount; i++) {
for (j=0; j<ColCount; j++)
printf("%7.4f ", pMatrix[i*ColCount+j]);
printf("\n");
}
}

// Function for matrix multiplication
```

```
void SerialResultCalculation(double* pAMatrix, double* pBMatrix,
double* pCMatrix, int Size) {
int i, j, k;  // Loop variablesfor
(i=0; i<Size; i++) {
for (j=0; j<Size; j++) for
(k=0; k<Size; k++)
pCMatrix[i*Size+j] += pAMatrix[i*Size+k]*pBMatrix[k*Size+j];
}
}

// Function for block multiplication
void BlockMultiplication(double* pAblock, double* pBblock,
double* pCblock, int Size) { SerialResultCalculation(pAblock,
pBblock, pCblock, Size);
}

// Creation of two-dimensional grid communicator
// and communicators for each row and each column of the gridvoid
CreateGridCommunicators() {
int DimSize[2];  // Number of processes in each dimension of the gridint
Periodic[2]; // =1, if the grid dimension should be periodic
int Subdims[2];  // =1, if the grid dimension should be fixed

DimSize[0]   =   GridSize;
DimSize[1]   =   GridSize;
Periodic[0] = 0;
Periodic[1] = 0;

// Creation of the Cartesian communicator MPI_Cart_create(MPI_COMM_WORLD,
2, DimSize, Periodic, 1, &GridComm);

// Determination of the cartesian coordinates for every process
MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);

// Creating communicators for rows
Subdims[0] = 0;  // Dimensionality fixing
Subdims[1] = 1;  // The presence of the given dimension in the subgrid
MPI_Cart_sub(GridComm, Subdims, &RowComm);

// Creating communicators for columnsSubdims[0] =
1;
Subdims[1] = 0;
MPI_Cart_sub(GridComm, Subdims, &ColComm);
}

// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
double* &pCMatrix, double* &pAblock, double* &pBblock, double* &pCblock,
double* &pTemporaryAblock, int &Size, int &BlockSize ) {
if (ProcRank == 0) {do
{
printf("\nEnter size of the initial objects: ");scanf("%d",
&Size);


if (Size%GridSize != 0) {
printf ("Size of matricies must be divisible by the grid size!\n");
}
}
while (Size%GridSize != 0);
}
MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

BlockSize = Size/GridSize;
```

```
pAblock  =  new  double  [BlockSize*BlockSize];
pBblock  =  new  double  [BlockSize*BlockSize];
pCblock = new double [BlockSize*BlockSize];
pTemporaryAblock = new double [BlockSize*BlockSize];

for (int i=0; i<BlockSize*BlockSize; i++) {
pCblock[i] = 0;
}
if (ProcRank == 0) {
pAMatrix   =   new   double   [Size*Size];
pBMatrix   =   new   double   [Size*Size];
pCMatrix = new double [Size*Size];
DummyDataInitialization(pAMatrix, pBMatrix, Size);
//RandomDataInitialization(pAMatrix, pBMatrix, Size);
}
}

// Function for checkerboard matrix decomposition
void CheckerboardMatrixScatter(double* pMatrix, double* pMatrixBlock,int
Size, int BlockSize) {
double * MatrixRow = new double [BlockSize*Size];if
(GridCoords[1] == 0) {
MPI_Scatter(pMatrix, BlockSize*Size, MPI_DOUBLE, MatrixRow,BlockSize*Size,
MPI_DOUBLE, 0, ColComm);
}

for (int i=0; i<BlockSize; i++) { MPI_Scatter(&MatrixRow[i*Size],
BlockSize, MPI_DOUBLE,
&(pMatrixBlock[i*BlockSize]), BlockSize, MPI_DOUBLE, 0, RowComm);
}
delete [] MatrixRow;
}

// Data distribution among the processes
void DataDistribution(double* pAMatrix, double* pBMatrix, double*
pMatrixAblock, double* pBblock, int Size, int BlockSize) {
// Scatter the matrix among the processes of the first grid column
CheckerboardMatrixScatter(pAMatrix, pMatrixAblock, Size, BlockSize);
CheckerboardMatrixScatter(pBMatrix, pBblock, Size, BlockSize);
}

// Function for gathering the result matrix
void ResultCollection (double* pCMatrix, double* pCblock, int Size,
int BlockSize) {
double * pResultRow = new double [Size*BlockSize];for
(int i=0; i<BlockSize; i++) {
MPI_Gather( &pCblock[i*BlockSize], BlockSize, MPI_DOUBLE,&pResultRow[i*Size],
BlockSize, MPI_DOUBLE, 0, RowComm);
}

if (GridCoords[1] == 0) {
MPI_Gather(pResultRow, BlockSize*Size, MPI_DOUBLE, pCMatrix,BlockSize*Size,
MPI_DOUBLE, 0, ColComm);


}
delete [] pResultRow;
}

// Broadcasting matrix A blocks to process grid rows
void ABlockCommunication (int iter, double *pAblock, double* pMatrixAblock,
int BlockSize) {
```

```
// Defining the leading process of the process grid rowint
Pivot = (GridCoords[0] + iter) % GridSize;

// Copying the transmitted block in a separate memory bufferif
(GridCoords[1] == Pivot) {
for (int i=0; i<BlockSize*BlockSize; i++)
pAblock[i] = pMatrixAblock[i];
}

// Block broadcasting
MPI_Bcast(pAblock, BlockSize*BlockSize, MPI_DOUBLE, Pivot, RowComm);
}

// Cyclic shift of matrix B blocks in the process grid columnsvoid
BblockCommunication (double *pBblock, int BlockSize) {
MPI_Status Status;
int NextProc = GridCoords[0] + 1;
if ( GridCoords[0] == GridSize-1 ) NextProc = 0;int
PrevProc = GridCoords[0] - 1;
if ( GridCoords[0] == 0 ) PrevProc = GridSize-1;

MPI_Sendrecv_replace( pBblock, BlockSize*BlockSize, MPI_DOUBLE,
NextProc, 0, PrevProc, 0, ColComm, &Status);
}

void ParallelResultCalculation(double* pAblock, double* pMatrixAblock,double*
pBblock, double* pCblock, int BlockSize) {
for (int iter = 0; iter < GridSize; iter ++) {
// Sending blocks of matrix A to the process grid rows ABlockCommunication
(iter, pAblock, pMatrixAblock, BlockSize);
// Block multiplication
BlockMultiplication(pAblock, pBblock, pCblock, BlockSize);
// Cyclic shift of blocks of matrix B in process grid columns
BblockCommunication(pBblock, BlockSize);
}
}

// Test printing of the matrix block
void TestBlocks (double* pBlock, int BlockSize, char str[]) {
MPI_Barrier(MPI_COMM_WORLD);
if (ProcRank == 0) {
printf("%s \n", str);
}
for (int i=0; i<ProcNum; i++) {if
(ProcRank == i) {
printf ("ProcRank = %d \n", ProcRank);
PrintMatrix(pBlock, BlockSize, BlockSize);
}
MPI_Barrier(MPI_COMM_WORLD);
}
}

void TestResult(double* pAMatrix, double* pBMatrix, double* pCMatrix,int
Size) {
double* pSerialResult;         // Result matrix of serial multiplication
double Accuracy = 1.e-6;       // Comparison accuracy
int equal = 0;                 // =1, if the matrices are not equal
int i;                         // Loop variable

if (ProcRank == 0) {
pSerialResult = new double [Size*Size];for
(i=0; i<Size*Size; i++) {
pSerialResult[i] = 0;
```

```
}
BlockMultiplication(pAMatrix, pBMatrix, pSerialResult, Size);for
(i=0; i<Size*Size; i++) {
if (fabs(pSerialResult[i]-pCMatrix[i]) >= Accuracy)equal =
1;
}
if (equal == 1)
printf("The results of serial and parallel algorithms are NOT "
"identical. Check your code.");
else
printf("The results of serial and parallel algorithms are "
"identical.");
}
}

// Function for computational process termination
void ProcessTermination (double* pAMatrix, double* pBMatrix,
double* pCMatrix, double* pAblock, double* pBblock, double* pCblock,double*
pMatrixAblock) {
if (ProcRank == 0) {
delete [] pAMatrix;
delete [] pBMatrix;
delete [] pCMatrix;
}
delete [] pAblock; delete
[] pBblock; delete []
pCblock; delete []
pMatrixAblock;
}

void main(int argc, char* argv[]) {
double* pAMatrix; // The first argument of matrix multiplicationdouble*
pBMatrix;  // The second argument of matrix multiplicationdouble*
pCMatrix;  // The result matrix
int Size;              // Size of matricies
int BlockSize;         // Sizes of matrix blocks on current process
double *pAblock;       // Initial block of matrix A on current
processdouble *pBblock; // Initial block of matrix B on current
processdouble *pCblock; // Block of result matrix C on current
process double *pMatrixAblock;
double Start, Finish, Duration;

setvbuf(stdout, 0, _IONBF, 0);

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

GridSize = sqrt((double)ProcNum); if
(ProcNum != GridSize*GridSize) {
if (ProcRank == 0) {
printf ("Number of processes must be a perfect square \n");
}
}
else {
    if (ProcRank == 0)
      printf("Parallel matrix multiplication program\n");


    // Creating the cartesian grid, row and column communcators
    CreateGridCommunicators();
```

```
    // Memory allocation and initialization of matrix elements
    ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,

      pCblock, pMatrixAblock, Size, BlockSize );



    DataDistribution(pAMatrix, pBMatrix, pMatrixAblock, pBblock, Size,
      BlockSize);


    // Execution of Fox method ParallelResultCalculation(pAblock,
    pMatrixAblock, pBblock,

      pCblock, BlockSize);


    ResultCollection(pCMatrix, pCblock, Size, BlockSize);TestResult(pAMatrix,

    pBMatrix, pCMatrix, Size);


    // Process Termination

    ProcessTermination (pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,pCblock,
      pMatrixAblock);

  }

  MPI_Finalize();
```

Лабораторная работа №3: «Параллельные методы решения систем линейных уравнений»

Системы линейных уравнений возникают при решении ряда прикладных задач, описываемых дифференциальными, интегральными или системами нелинейных (трансцендентных) уравнений. Они могут появляться также в задачах математического программирования, статистической обработки данных, аппроксимации функций, при дискретизации краевых дифференциальных задач методом конечных разностей или методом конечных элементов и др.

В данной лабораторной работе рассматривается один из прямых методов решения систем линейных уравнений – метод Гаусса и его параллельное обобщение.

Целью данной лабораторной работы является разработка параллельной программы, которая выполняет решение системы линейных уравнений методом Гаусса. Выполнение лабораторной включает выполнение следующих задач:

1. Определение задачи решения системы линейных уравнений
2. Изучение последовательного алгоритма Гаусса решения систем линейных уравнений
3. Реализация последовательного алгоритма Гаусса
4. Разработка параллельного алгоритма Гаусса
5. Реализация параллельного алгоритма Гаусса решения систем линейных уравнений

Задача 1. **Определение задачи решения системы линейных уравнений**

Линейное уравнение с n неизвестными $x_0, x_1, \ldots, x_{n-1}$ может быть определено при помощи

выражения $a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} = b$, где величины $a_0, a_1, ..., a_{n-1}$ и b представляют собой постоянные значения.

Система линейных уравнений или *линейная система в матричном* виде может представлена как

$$Ax = b \,,$$

где $A=(a_{i,j})$ есть вещественная матрица размера $n \times n$, а вектора b и x состоят из n элементов.

Под *задачей решения системы линейных уравнений* для заданных матрицы A и вектора b обычно понимается нахождение значения вектора неизвестных x, при котором выполняются все уравнения системы.

Задача 2. Изучение последовательного алгоритма Гаусса решения систем линейных уравнений.

Метод Гаусса является широко известным *прямым* алгоритмом решения систем линейных уравнений, для которых матрицы коэффициентов являются *плотными*. Если система линейных уравнений является *невырожденной*, то метод Гаусса гарантирует нахождение решения с погрешностью, определяемой точностью машинных вычислений. Основная идея метода состоит в приведении матрицы A посредством эквивалентных преобразований (не меняющих решение системы) к треугольному виду, после чего значения искомых неизвестных может быть получено непосредственно в явном виде.

В **задаче 2** дается общая характеристика метода Гаусса, достаточная для начального понимания алгоритма и позволяющая рассмотреть возможные способы параллельных вычислений при решении систем линейных уравнений.

Метод Гаусса основывается на возможности выполнения преобразований линейных уравнений, которые не меняют при этом решение рассматриваемой системы (такие преобразования носят наименование *эквивалентных*). К числу таких преобразований относятся:

- Умножение любого из уравнений на ненулевую константу,
- Перестановка уравнений,
- Прибавление к уравнению любого другого уравнения системы.

Метод Гаусса включает последовательное выполнение двух этапов. На первом этапе – *прямой ход* метода Гаусса – исходная система линейных уравнений при помощи последовательного исключения неизвестных приводится к верхнему треугольному виду.

На *обратном ходе* метода Гаусса (второй этап алгоритма) осуществляется определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_{n-1} , после этого из предпоследнего уравнения становится возможным определение переменной x_{n-2} и т.д.

При выполнении **задачи 3** необходимо реализовать последовательный алгоритм Гаусса решения систем линейных уравнений. Начальный вариант будущей программы представлен. Он содержит часть исходного кода и в котором заданы необходимые параметры проекта. В ходе выполнения задания необходимо дополнить имеющийся вариант программы операциями ввода размера матриц, задание исходных данных, реализации алгоритма Гаусса и вывода результатов.

Программный код последовательного алгоритмаГаусса решения линейных

систем

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>


int* pSerialPivotPos;      //  The  Number  of  pivot  rows  selected  at
theiterations

int* pSerialPivotIter; // The Iterations, at which the rows were pivots


// Function for simple initialization of the matrix and the vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
    int i, j;  // Loop variables


    for (i=0; i<Size; i++)
      {pVector[i] = i+1;
        for (j=0; j<Size; j++)
        {if (j <= i)
            pMatrix[i*Size+j] =
          1;else
            pMatrix[i*Size+j] = 0;
        }
    }
}


// Function for random initialization of the matrix and the vector elements
void RandomDataInitialization (double* pMatrix, double* pVector, int Size)
{
    int i, j;  // Loop
    variables
    srand(unsigned(clock()));
    for (i=0; i<Size; i++) {
      pVector[i] =
        rand()/double(1000);for (j=0;
        j<Size; j++) {
          if (j <= i)
          pMatrix[i*Size+j] =
          rand()/double(1000);else
          pMatrix[i*Size+j] = 0;
        }
    }
}


// Function for memory allocation and definition of the objects elements
void ProcessInitialization (double* &pMatrix, double* &pVector,
      double* &pResult, int &Size) {
    // Setting the size of the matrix and the
    vectordo {
```

```
        printf("\nEnter size of the matrix and the vector: ");
        scanf("%d", &Size);

        printf("\nChosen size = %d \n", Size);


          if (Size <= 0)

          printf("\nSize of objects must be greater than 0!\n");
      } while (Size <= 0);


      // Memory allocation

      pMatrix = new double
      [Size*Size];pVector = new double
      [Size]; pResult = new double
      [Size];


      // Initialization of the matrix and the vector elements


      DummyDataInitialization(pMatrix, pVector, Size);

      //RandomDataInitialization(pMatrix, pVector, Size);
}


// Function for formatted matrix output
  void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
      int i, j; // Loop variables

      for (i=0; i<RowCount; i++)
        {for (j=0; j<ColCount;
        j++)
            printf("%7.4f ",
          pMatrix[i*RowCount+j]);printf("\n");

      }
}


// Function for formatted vector output
  void PrintVector (double* pVector, int Size) {
      int i;
      for (i=0; i<Size; i++)
        printf("%7.4f ", pVector[i]);

}


// Finding the pivot row
  int FindPivotRow(double* pMatrix, int Size, int Iter) {
      int PivotRow = -1; // The index of the pivot row

      int MaxValue =  0; // The value of the pivot
      elementint i;      // Loop variable


      // Choose the row, that stores the maximum element
      for (i=0; i<Size; i++) {

        if ((pSerialPivotIter[i] == -1) &&
           (fabs(pMatrix[i*Size+Iter]) > MaxValue))
```

```
        {
        PivotRow = i;
        MaxValue = fabs(pMatrix[i*Size+Iter]);
      }
    }
    return PivotRow;
}

// Column elimination
  void SerialColumnElimination (double* pMatrix, double* pVector, int Pivot,
    int Iter, int Size) {
    double PivotValue, PivotFactor;
    PivotValue =
    pMatrix[Pivot*Size+Iter];for (int
    i=0; i<Size; i++) {
      if (pSerialPivotIter[i] == -1) {
        PivotFactor = pMatrix[i*Size+Iter] /
        PivotValue;for (int j=Iter; j<Size; j++) {
          pMatrix[i*Size + j] -= PivotFactor * pMatrix[Pivot*Size+j];
        }
        pVector[i] -= PivotFactor * pVector[Pivot];
      }
    }
}

// Gaussian elimination
  void SerialGaussianElimination(double* pMatrix,double* pVector,int Size)
    {int Iter;      // The number of the iteration of the gaussian
                    // elimination
    int PivotRow;   // The number of the current pivot
    rowfor (Iter=0; Iter<Size; Iter++) {
      // Finding the pivot row
      PivotRow = FindPivotRow(pMatrix, Size, Iter);
      pSerialPivotPos[Iter] = PivotRow;

      pSerialPivotIter[PivotRow] = Iter;
      SerialColumnElimination(pMatrix, pVector, PivotRow, Iter,
      Size);
    }
}

// Back substution
  void SerialBackSubstitution (double* pMatrix, double* pVector,
    double* pResult, int Size) {
    int RowIndex, Row;
    for (int i=Size-1; i>=0; i--) {
      RowIndex = pSerialPivotPos[i];
      pResult[i] = pVector[RowIndex]/pMatrix[Size*RowIndex+i];
```

```
      for (int j=0; j<i; j++) {
        Row = pSerialPivotPos[j];
        pVector[j] -= pMatrix[Row*Size+i]*pResult[i];
        pMatrix[Row*Size+i] = 0;
      }
    }
}
// Function for the execution of Gauss algorithm
  void SerialResultCalculation(double* pMatrix, double* pVector,
    double* pResult, int Size) {

    // Memory allocation
    pSerialPivotPos  = new int
    [Size];pSerialPivotIter = new
    int [Size];for (int i=0; i<Size;
    i++) {
      pSerialPivotIter[i] = -1;
    }
    // Gaussian elimination
    SerialGaussianElimination (pMatrix, pVector, Size);
    // Back substitution
    SerialBackSubstitution (pMatrix, pVector, pResult, Size);


    // Memory deallocation
    delete []
    pSerialPivotPos; delete
    [] pSerialPivotIter;
}

// Function for computational process termination
  void ProcessTermination (double* pMatrix, double* pVector, double* pResult)
{
    delete         []
    pMatrix;  delete
    []      pVector;
    delete         []
    pResult;
}
  void main() {
    double* pMatrix; // The matrix of the linear system
    double* pVector;  // The right parts of the linear
    systemdouble* pResult;  // The result vector
    int Size;         // The sizes of the initial matrix and the
    vectortime_t start, finish;
    double duration;


    printf("Serial Gauss algorithm for solving linear systems\n");
    // Memory allocation and definition of objects' elements
    ProcessInitialization(pMatrix, pVector, pResult, Size);
```

```
    // The matrix and the vector output
    printf ("Initial Matrix \n");
    PrintMatrix(pMatrix, Size, Size);
    printf("Initial Vector \n");
    PrintVector(pVector, Size);


    // Execution of Gauss algorithm
    start = clock();

    SerialResultCalculation(pMatrix, pVector, pResult, Size);finish
    = clock();

    duration = (finish-start)/double(CLOCKS_PER_SEC);


    // Printing the result vector
    printf ("\n Result Vector: \n");
    PrintVector(pResult, Size);


    // Printing the execution time of Gauss method
    printf("\n Time of execution: %f\n", duration);


    // Computational process termination
    ProcessTermination(pMatrix, pVector, pResult);
    getch();
}
```

Задача 4. Разработка параллельного алгоритма Гаусса

При внимательном рассмотрении метода Гаусса можно заметить, что все вычисления сводятся к однотипным вычислительным операциям над строками матрицы коэффициентов системы линейных уравнений. Как результат, в основу параллельной реализации алгоритма Гаусса может быть положен принцип распараллеливания по данным. В качестве *базовой подзадачи* можно принять тогда все вычисления, связанные с обработкой одной строки матрицы A и соответствующего элемента вектора b.

Выделение информационных зависимостей

Рассмотрим общую схему параллельных вычислений и возникающие при этом информационные зависимости между базовыми подзадачами.

Для выполнения **прямого хода** метода Гаусса необходимо осуществить $(n-1)$ итерацию по исключению неизвестных для преобразования матрицы коэффициентов A к верхнему треугольномувиду.

Выполнение итерации i, $0 \le i<n-1$, прямого хода метода Гаусса включает ряд последовательных действий. Прежде всего, в самом начале итерации необходимо выбрать ведущую строку, которая при использовании метода главных элементов определяется поиском строки с наибольшим по абсолютной величине значением среди элементов столбца i , соответствующего исключаемой переменной x_i . Поскольку строки матрицы A распределены по подзадачам, для поиска максимального значения подзадачи с номерами k, $k>i$, должны обменяться своими элементами при исключаемой переменной x_i. После сбора всех необходимых данных в каждой подзадаче может быть определено, какая из подзадач содержит ведущую строку и какое значение является ведущим элементом.

Далее для продолжения вычислений ведущая подзадача должна разослать свою строку матрицы A и соответствующий элемент вектора b всем остальным подзадачам с номерами k, $k>i$. Получив ведущую строку, подзадачи выполняют вычитание строк, обеспечивая тем самым исключение соответствующей неизвестной x_i.

При выполнении **обратного хода** метода Гаусса подзадачи выполняют необходимые вычисления для нахождения значения неизвестных. Как только какая-либо подзадача i, $0 \le i<n-1$, определяет значение своей переменной x_i , это значение должно быть разослано всем подзадачам с номерами k, $k<i$. Далее подзадачи подставляют полученное значение новой неизвестной и выполняют корректировку значений для элементов вектора b.

Масштабирование и распределение подзадач по процессорам

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и сбалансированным объемом передаваемых данных. В случае, когда размер матрицы, описывающей систему линейных уравнений, оказывается большим, чем число доступных процессоров (т.е., $p<n$), базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько строк матрицы. Воспользуемся уже знакомой ленточной схемой разделения данных: каждому процессу выделяется непрерывная последовательность строк матрицы линейных уравнений.

Распределение подзадач между процессорами должно учитывать характер выполняемых в методе Гаусса коммуникационных операций. Основным видом информационного взаимодействия подзадач является операция передачи данных от одного процессора всем процессорам вычислительной системы. Как результат, для эффективной реализации требуемых информационных взаимодействий между базовыми подзадачами топология сети передачи данных должны иметь структуру гиперкуба или полногографа.

Задача 5 - Реализация параллельного алгоритма Гаусса решения систем линейных уравнений

При выполнении задания необходимо разработать параллельный алгоритм Гаусса для решения систем линейных уравнений.

Программный код параллельного алгоритма Гаусса решения линейных систем

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>
#include <mpi.h>

int ProcNum;             // Number of the available processes

int ProcRank;            // Rank of the current process

int *pParallelPivotPos; // Number of rows selected as the pivot ones
int *pProcPivotIter;     // Number of iterations, at which the
processor
                         // rows were used as the pivot ones
```

```
int* pProcInd; // Number of the first row located on the processes
int* pProcNum; // Number of the linear system rows located on the processes

// Function for simple definition of matrix and vector elements
  void DummyDataInitialization (double* pMatrix, double* pVector, int Size)
    {int i, j;  // Loop variables

    for (i=0; i<Size; i++)
      {pVector[i] = i+1;
        for (j=0; j<Size; j++)
        {if (j <= i)
            pMatrix[i*Size+j] =
          1;else
            pMatrix[i*Size+j] = 0;
        }
    }
}

// Function for random definition of matrix and vector elements
  void RandomDataInitialization(double* pMatrix, double* pVector, int Size)
    {int i, j;  // Loop variables
    srand(unsigned(clock()));

    for (i=0; i<Size; i++) {
      pVector[i] =
      rand()/double(1000);
        for (j=0; j<Size; j++)
          {if (j <= i)
          pMatrix[i*Size+j] =
          rand()/double(1000);else
          pMatrix[i*Size+j] = 0;
        }
    }
}

// Function for memory allocation and data initialization
  void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, double* &pProcRows, double* &pProcVector,
    double* &pProcResult, int &Size, int &RowNum) {

    int RestRows; // Number of rows, that haven't been distributed
    yetint i;     // Loop variable

    if (ProcRank == 0)
      {do {
        printf("\nEnter the size of the matrix and the vector: ");
        scanf("%d", &Size);
        if (Size < ProcNum) {
```

```
            printf("Size must be greater than number of processes! \n");
          }
        }
        while (Size < ProcNum);
      }
      MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

      RestRows = Size;
      for (i=0; i<ProcRank; i++)
        RestRows = RestRows-RestRows/(ProcNum-i);
      RowNum = RestRows/(ProcNum-ProcRank);

      pProcRows = new double [RowNum*Size];
      pProcVector = new double [RowNum];
      pProcResult = new double [RowNum];

      pParallelPivotPos = new int [Size];
      pProcPivotIter = new int [RowNum];

      pProcInd = new int
      [ProcNum];pProcNum = new int
      [ProcNum];

      for (int i=0; i<RowNum;
        i++)pProcPivotIter[i] =
        -1;

      if (ProcRank == 0) {
        pMatrix = new double
        [Size*Size];pVector = new
        double [Size]; pResult = new
        double [Size];
          // DummyDataInitialization (pMatrix, pVector, Size);
        RandomDataInitialization(pMatrix, pVector, Size);
      }
}

// Function for the data distribution among the processes
  void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
    double* pProcVector, int Size, int RowNum) {

    int *pSendNum;      // Number of the elements sent to the process

    int *pSendInd;      // Index of the first data element sent
                        // to the process
    int RestRows=Size;  // Number of rows, that have not been
                        // distributed yet
```

```
int i;             // Loop variable

  // Alloc memory for temporary
  objectspSendInd = new int
  [ProcNum]; pSendNum = new int
  [ProcNum];

  // Define the disposition of the matrix rows for the current process
  RowNum = (Size/ProcNum);
  pSendNum[0] =
  RowNum*Size;pSendInd[0]
  = 0;
  for (i=1; i<ProcNum; i++)
    {RestRows -= RowNum;
    RowNum = RestRows/(ProcNum-i);
    pSendNum[i] = RowNum*Size;
    pSendInd[i] = pSendInd[i-1]+pSendNum[i-1];
  }

  // Scatter the rows
  MPI_Scatterv(pMatrix, pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Define the disposition of the matrix rows for current process
  RestRows = Size;
  pProcInd[0] = 0;
  pProcNum[0] =
  Size/ProcNum;for (i=1;
  i<ProcNum; i++) {
    RestRows -= pProcNum[i-1];
    pProcNum[i] = RestRows/(ProcNum-i);
    pProcInd[i] = pProcInd[i-1]+pProcNum[i-1];
  }

  MPI_Scatterv(pVector, pProcNum, pProcInd, MPI_DOUBLE, pProcVector,
    pProcNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Free the
  memory delete []
  pSendNum;delete
  [] pSendInd;
}

// Function for gathering the result vector
 void ResultCollection(double* pProcResult, double* pResult) {
   //Gather the whole result vector on every processor
  MPI_Gatherv(pProcResult, pProcNum[ProcRank], MPI_DOUBLE, pResult,
    pProcNum, pProcInd, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

```
// Function for formatted matrix output
  void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
    int i, j; // Loop variables

    for (i=0; i<RowCount; i++)
      {for (j=0; j<ColCount;
      j++)

          printf("%7.4f ",
        pMatrix[i*ColCount+j]);printf("\n");

    }

}


// Function for formatted vector output
  void PrintVector (double* pVector, int Size) {
    int i;


    for (i=0; i<Size; i++)
      printf("%7.4f ", pVector[i]);

}


// Function for formatted result vector output
  void PrintResultVector (double* pResult, int Size) {
    int i;
    for (i=0; i<Size; i++)

          printf("%7.4f ", pResult[pParallelPivotPos[i]]);

}


// Fuction for the column elimination
  void ParallelEliminateColumns(double* pProcRows, double* pProcVector,
    double* pPivotRow, int Size, int RowNum, int Iter) {

    double multiplier;

    for (int i=0; i<RowNum; i++) {
      if (pProcPivotIter[i] == -1)
      {

        multiplier = pProcRows[i*Size+Iter] / pPivotRow[Iter];
        for (int j=Iter; j<Size; j++) {

          pProcRows[i*Size + j] -= pPivotRow[j]*multiplier;

        }

        pProcVector[i] -= pPivotRow[Size]*multiplier;

      }

    }

}


// Function for the Gausian elimination
  void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
    int Size, int RowNum) {
    double MaxValue;   // Value of the pivot element of the process

    int    PivotPos;   // Position of the pivot row in the process stripe
```

```
// Structure for the pivot row selection
struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;

// pPivotRow is used for storing the pivot row and the corresponding
// element of the vector b
double* pPivotRow = new double [Size+1];

// The iterations of the Gaussian elimination
stagefor (int i=0; i<Size; i++) {

  // Calculating the local pivot row
  double MaxValue = 0;
    for (int j=0; j<RowNum; j++) {
    if ((pProcPivotIter[j] == -1)
    &&
        (MaxValue < fabs(pProcRows[j*Size+i]))) {
      MaxValue = fabs(pProcRows[j*Size+i]);
      PivotPos = j;
      }
  }
  ProcPivot.MaxValue = MaxValue;
  ProcPivot.ProcRank = ProcRank;

  // Find the pivot process (process with the maximum value of MaxValue)
  MPI_Allreduce(&ProcPivot, &Pivot, 1, MPI_DOUBLE_INT, MPI_MAXLOC,
                MPI_COMM_WORLD);

  // Broadcasting the pivot row
  if ( ProcRank == Pivot.ProcRank ){
    pProcPivotIter[PivotPos]= i; //iteration number
    pParallelPivotPos[i]= pProcInd[ProcRank] + PivotPos;
    }
  MPI_Bcast(&pParallelPivotPos[i], 1, MPI_INT, Pivot.ProcRank,

    MPI_COMM_WORLD);

  if ( ProcRank == Pivot.ProcRank ){
    // Fill the pivot row
    for (int j=0; j<Size; j++) {
      pPivotRow[j] = pProcRows[PivotPos*Size + j];
    }
    pPivotRow[Size] = pProcVector[PivotPos];
  }
  MPI_Bcast(pPivotRow, Size+1, MPI_DOUBLE, Pivot.ProcRank,
    MPI_COMM_WORLD);
```

```
      ParallelEliminateColumns(pProcRows, pProcVector, pPivotRow, Size,
        RowNum, i);
    }
}
// Function for finding the pivot row of the back substitution
void FindBackPivotRow (int RowIndex, int Size, int &IterProcRank,
    int &IterPivotPos) {
    for (int i=0; i<ProcNum-1; i++) {
      if ((pProcInd[i]<=RowIndex) && (RowIndex<pProcInd[i+1]))
          IterProcRank = i;
    }
    if (RowIndex >= pProcInd[ProcNum-1])
      IterProcRank = ProcNum-1;
    IterPivotPos = RowIndex - pProcInd[IterProcRank];
}

// Function for the back substitution
  void ParallelBackSubstitution (double* pProcRows, double* pProcVector,
    double* pProcResult, int Size, int RowNum) {
    int IterProcRank;    // Rank of the process with the current pivot
    rowint IterPivotPos; // Position of the pivot row of the process
    double IterResult;   // Calculated value of the current unknown
    double val;

    // Iterations of the back substitution stage
    for (int i=Size-1; i>=0; i--) {

      // Calculating the rank of the process, which holds the pivot row
        FindBackPivotRow(pParallelPivotPos[i], Size, IterProcRank,
          IterPivotPos);

      // Calculating the unknown
      if (ProcRank == IterProcRank) {
        IterResult =
          pProcVector[IterPivotPos]/pProcRows[IterPivotPos*Size+i];
        pProcResult[IterPivotPos] = IterResult;
      }
      // Broadcasting the value of the current unknown
      MPI_Bcast(&IterResult, 1, MPI_DOUBLE, IterProcRank, MPI_COMM_WORLD);

      // Updating the values of the vector b
      for (int j=0; j<RowNum; j++)
        if ( pProcPivotIter[j] < i ) {
          val = pProcRows[j*Size + i] * IterResult;
          pProcVector[j]=pProcVector[j] - val;
      }
    }
```

```
}

  void TestDistribution(double* pMatrix, double* pVector, double* pProcRows,
    double* pProcVector, int Size, int RowNum) {


    if (ProcRank == 0) {
      printf("Initial Matrix: \n");
      PrintMatrix(pMatrix, Size,
      Size);printf("Initial Vector:
      \n"); PrintVector(pVector,
      Size);

    }

    MPI_Barrier(MPI_COMM_WORLD);
    for (int i=0; i<ProcNum; i++)
    {

      if (ProcRank == i) {

        printf("\nProcRank = %d \n", ProcRank);
        printf(" Matrix Stripe:\n");
        PrintMatrix(pProcRows, RowNum, Size);
        printf(" Vector: \n");
        PrintVector(pProcVector, RowNum);

      }

      MPI_Barrier(MPI_COMM_WORLD);

    }

}

// Function for the execution of the parallel Gauss algorithm

  void ParallelResultCalculation(double* pProcRows, double* pProcVector,
    double* pProcResult, int Size, int RowNum) {
    ParallelGaussianElimination (pProcRows, pProcVector, Size, RowNum);
    ParallelBackSubstitution (pProcRows, pProcVector, pProcResult, Size,

      RowNum);

}


// Function for computational process termination

  void ProcessTermination (double* pMatrix, double* pVector, double*
    pResult,double* pProcRows, double* pProcVector, double* pProcResult) {

    if (ProcRank == 0)
      {   delete    []
      pMatrix;  delete
      []      pVector;
      delete        []
      pResult;

    }

    delete [] pProcRows;
    delete []
    pProcVector;delete
    [] pProcResult;


    delete []
    pParallelPivotPos;delete
    [] pProcPivotIter;
```

```
    delete []
    pProcInd;delete
    [] pProcNum;

}


// Function for testing the result
void TestResult(double* pMatrix, double* pVector, double* pResult, int
Size) {

    /* Buffer for storing the vector, that is a result of multiplication
       of the linear system matrix by the vector of unknowns */

    double* pRightPartVector;

    // Flag, that shows wheather the right parts vectors are identical or
    notint equal = 0;

    double Accuracy = 1.e-6; // Comparison accuracy


    if (ProcRank == 0) {

      pRightPartVector = new double [Size];
        for (int i=0; i<Size; i++) {

          pRightPartVector[i] = 0;

          for (int j=0; j<Size; j++)
            {pRightPartVector[i] +=

              pMatrix[i*Size+j]*pResult[pParallelPivotPos[j]];

          }

        }

      for (int i=0; i<Size; i++) {

          if (fabs(pRightPartVector[i]-pVector[i]) > Accuracy))
          equal = 1;

      }

      if (equal == 1)

        printf("The result of the parallel Gauss algorithm is NOT correct."
               "Check your code.");

      else

        printf("The result of the parallel Gauss algorithm is
      correct.");delete [] pRightPartVector;

    }

}


void main(int argc, char* argv[]) {

    double* pMatrix;         // Matrix of the linear system
    double* pVector;         // Right parts of the linear
    systemdouble* pResult;   // Result vector

    double *pProcRows;       // Rows of the matrix
    A double *pProcVector;   // Block of the vector
    bdouble *pProcResult;    // Block of the vector
    x

    int     Size;            // Size of the matrix and
    vectorsint               RowNum; // Number of the matrix
```

```
rows double start, finish, duration;

setvbuf(stdout, 0, _IONBF,

0);MPI_Init ( &argc, &argv

);

MPI_Comm_rank ( MPI_COMM_WORLD, &ProcRank );
MPI_Comm_size ( MPI_COMM_WORLD, &ProcNum );

if (ProcRank == 0)
      printf("Parallel Gauss algorithm for solving linear systems\n");

// Memory allocation and data initialization
ProcessInitialization(pMatrix, pVector, pResult,

  pProcRows, pProcVector, pProcResult, Size, RowNum);

// The execution of the parallel Gauss
algorithmstart = MPI_Wtime();

DataDistribution(pMatrix, pProcRows, pVector, pProcVector, Size, RowNum);

ParallelResultCalculation(pProcRows, pProcVector, pProcResult, Size,

  RowNum);

TestDistribution(pMatrix, pVector, pProcRows, pProcVector, Size, RowNum);

ResultCollection(pProcResult, pResult);

finish = MPI_Wtime();
duration = finish-
start;

if (ProcRank == 0) {

  // Printing the result vector
  printf ("\n Result Vector:
  \n");
  PrintResultVector(pResult,
  Size);

}

TestResult(pMatrix, pVector, pResult, Size);

// Printing the time spent by Gauss algorithm
if (ProcRank == 0)

 printf("\n Time of execution: %f\n", duration);

// Computational process termination

ProcessTermination(pMatrix, pVector, pResult, pProcRows, pProcVector,
 pProcResult);

MPI_Finalize();

}
```

Лабораторная работа №4: «Параллельные методы сортировки данных»

Целью данной лабораторной работы является разработка параллельной программы, которая выполняет сортировку данных. Выполнение лабораторной включает выполнение следующих задач:

1. Постановка задачи сортировки.
2. Реализация последовательного алгоритма сортировки.
3. Разработка параллельного алгоритма сортировки.
4. Реализация параллельного алгоритма сортировки.

При выполнении задачи 1 необходимо изучить последовательный алгоритм пузырьковой сортировки. В ходе выполнения задачи 1 необходимо изучить основные принципы, используемые при сортировке данных.

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений в порядке монотонного возрастания или убывания.

Вычислительная трудоемкость процедуры упорядочивания является достаточно высокой. Так, для ряда известных простых методов (пузырьковая сортировка, сортировка включением и др.) количество необходимых операций определяется квадратичной зависимостью от числа упорядочиваемых данных

$$T \sim n^2 .$$

Для более эффективных алгоритмов (сортировка слиянием, сортировка Шелла, быстраясортировка) трудоемкость определяется величиной

$$T \sim n \log_2 n .$$

Данное выражение дает также нижнюю оценку необходимого количества операций для упорядочивания набора из n значений; алгоритмы с меньшей трудоемкостью могут быть получены только для частных вариантов задачи.

Ускорение сортировки может быть обеспечено при использовании нескольких (p > 1) процессоров. Исходный упорядочиваемый набор в этом случае разделяется между процессорами; в ходе сортировки данные пересылаются между процессорами и сравниваются между собой. Результирующий (упорядоченный) набор, как правило, также разделен между процессорами; при этом для систематизациитакого разделения для процессоров вводится та или иная система последовательной нумерации и обычно требуется, чтобы при завершении сортировки значения, располагаемые на процессорах с меньшими номерами, не превышали значений процессоров с большими номерами.

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений

$$S = \{a_1, a_2, ..., a_n\}$$

в порядке монотонного возрастания или убывания

$$S \sim S' = \{(a_1', a_2', ..., a_n') : a_1' \le a_2' \le ... \le a_n'\}$$

При внимательном рассмотрении способов упорядочивания данных, применяемых в алгоритмах сортировки, можно обратить внимание, что многие методы основаны на применении одной и той же *базовой операции* "*сравнить и переставить*" (*compare-exchange*), состоящей в сравнении той или иной пары значений из сортируемого набора данных и перестановки этих значений, если их

```
// Basic compare-exchange operation
if(A[i] > A[j]) {
  temp = A[i];
  A[i] = A[j];
  A[j] = temp;
}
```

порядок не соответствует условиям сортировки.

Целенаправленное применение данной операции позволяет упорядочить данные; в способах выбора пар значений для сравнения собственно и проявляется различие алгоритмов сортировки.

Выберем для примера реализации в задаче 1, один из наиболее простых методов упорядочивания данных - *алгоритм пузырьковой сортировки.* Этот алгоритм сравнивает и обменивает соседние элементы в последовательности, которую нужно отсортировать. Для последовательности

$(a_1, a_2, \ldots, a_n)$

алгоритм сначала выполняет n -1 базовых операций "сравнения-обмена" для последовательных пар элементов

$(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)$.

В результате после первой итерации алгоритма самый большой элемент перемещается ("всплывает") в конец последовательности. Далее последний элемент в преобразованной последовательности может быть исключен из рассмотрения, и описанная выше процедура применяется к оставшейся части последовательности

$(a'_1, a'_2, \ldots, a'_{n-1})$.

Как можно увидеть, последовательность будет отсортирована после n -1 итерации. Эффективность пузырьковой сортировки может быть улучшена, если завершать алгоритм в случае отсутствия каких- либо изменений сортируемой

```
// Sequential bubble sorting algorithm
BubbleSort(double A[], int n) {
  for(i = 1; i < n; i++)
    for(j = 0; j < n - i; j++)
      compare exchange(A[j], A[j+1]);
```

последовательности данных в ходе какой-либо итерации сортировки.

При выполнении **задачи 2** необходимо реализовать последовательный алгоритм пузырьковой сортировки. Начальный вариант будущей программы представлен в проекте *SerialBubbleSort*, который содержит некоторую часть исходного кода. В ходе выполнения упражнения необходимо дополнить имеющийся вариант программы операциями ввода количества исходных данных, инициализации данных для сортировки, непосредственно сортировки данных и проверки правильности результатов работы программы.

SerialBubbleSort.cpp

```
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <ctime>
#include "SerialBubbleSort.h"
#include "SerialBubbleSortTest.h"

using namespace std;

const double RandomDataMultiplier = 1000.0;int

main(int argc, char *argv[]) {
  double *pData = 0;
  int DataSize = 0;
    time_t start, finish;
    double duration = 0.0;

    printf("Serial bubble sort program\n");

    // Process initialization
    ProcessInitialization(pData, DataSize);

    printf("Data before
    sorting\n");PrintData(pData,
    DataSize);

    // Serial bubble
    sortstart =
    clock();
    SerialBubble(pData,
    DataSize);finish = clock();

    printf("Data after
    sorting\n");PrintData(pData,
    DataSize);

    duration = (finish - start) /
    double(CLOCKS_PER_SEC);printf("Time of execution:
    %f\n", duration);

    // Process termination
    ProcessTermination(pData
    );

    return 0;
  }

  // Function for allocating the memory and setting the initial
  valuesvoid ProcessInitialization(double *&pData, int& DataSize) {
    do {
      printf("Enter the size of data to be sorted:
      ");scanf("%d", &DataSize);
      if(DataSize <= 0)
        printf("Data size should be greater than zero\n");
    }
    while(DataSize <= 0);

    printf("Sorting %d data items\n",

    DataSize);pData = new double[DataSize];
```

```
    // Simple setting the data
    DummyDataInitialization(pData, DataSize);

    // Setting the data by the random generator
    //RandomDataInitialization(pData, DataSize);
  }

  // Function for computational process
  terminationvoid ProcessTermination(double
  *pData) {
    delete []pData;
  }

  // Function for simple setting the initial data
  void DummyDataInitialization(double*& pData, int& DataSize)
    {for(int i = 0; i < DataSize; i++)
      pData[i] = DataSize - i;
  }

  // Function for initializing the data by the random generator
  void RandomDataInitialization(double *&pData, int& DataSize) {

  srand( (unsigned)time(0) );

  for(int i = 0; i < DataSize; i++)
    pData[i] = double(rand()) / RAND_MAX * RandomDataMultiplier;
}

// Serial bubble sort algorithm
void SerialBubble(double *pData, int DataSize) {
  double Tmp;

  for(int i = 1; i < DataSize; i++)
    for(int j = 0; j < DataSize - i; j++)
      if(pData[j] > pData[j + 1]) {
        Tmp          = pData[j];
        pData[j]     = pData[j + 1];
        pData[j + 1] = Tmp;
      }
}
```

В файле **SerialBubbleSortTest.cpp** содержатся подготовленные варианты тестовых функций, которые понадобятся для проверки правильности разрабатываемых функций.

Файл SerialBubbleSortTest.cpp

```
#include <algorithm>
#include <cstdio>
using namespace std;

// Function for formatted data output
void PrintData(double *pData, int DataSize) {
  for(int i = 0; i < DataSize; i++)
    printf("%7.4f ", pData[i]);
  printf("\n");
}

// Sorting by the standard library algorithm
void SerialStdSort(double *pData, int DataSize) {
  sort(pData, pData + DataSize);
}
```

При выполнении **задачи 3** необходимо изучить принципы распараллеливания алгоритма пузырьковой сортировки. Для этого необходимо провести декомпозицию задачи, выделить информационные взаимодействия между подзадачами, определить вычислительную схему и выполнить распределение набора подзадач по вычислительным устройствам.

Принципы распараллеливания

Алгоритм пузырьковой сортировки в прямом виде достаточно сложен для распараллеливания – сравнение пар значений упорядочиваемого набора данных происходит строго последовательно. В связи с этим для параллельного применения используется не сам этот алгоритм, а его модификация, известная в литературе как метод *чет-нечетной перестановки* (*odd-even transposition*) . Суть модификации состоит в том, что в алгоритм сортировки вводятся два разных правила выполнения итераций метода – в зависимости от четности или нечетности номера итерации сортировки для обработки выбираются элементы с четными или нечетными индексами соответственно, сравнение выделяемых значений всегда осуществляется с их правыми соседними элементами. Таким образом, на всех нечетных итерациях сравниваются пары

$(a_1, a_2), (a_3, a_4), ..., (a_{n-1}, a_n)$ (при четном n),

а на четных итерациях обрабатываются элементы

$(a_2, a_3), (a_4, a_5), ..., (a_{n-2}, a_{n-1})$.

После n - кратного повторения итераций сортировки исходный набор данных оказывается упорядоченным.

```
// Sequential odd-even transposition algorithm
OddEvenSort ( double A[], int n ) {
  for ( i=1; i<n; i++ ) {
    if ( i%2==1 ) { // odd iteration
      for ( j=0; j<n/2-2; j++ )
        compare_exchange(A[2j+1],A[2j+2]);
      if ( n%2==1 ) // the comparison of the last pair, if n is odd
        compare_exchange(A[n-2],A[n-1]);
     }
     if ( i%2==0 )     // even iteration
       for ( j=1; j<n/2-1; j++ )
```

Определение подзадач и выделение информационных зависимостей

Получение параллельного варианта для метода чет-нечетной перестановки уже не представляет каких-либо затруднений – сравнения пар значений на итерациях сортировки являются независимыми и могут быть выполнены параллельно. В случае $p<n$, когда количество процессоров является меньшим числа упорядочиваемых значений, процессоры содержат блоки данных размера n/p . Алгоритм сортировки в этом случае может быть получен как обобщение процедуры чет-нечетной сортировки.

Следуя схеме одноэлементного сравнения, взаимодействие пары процессоров P_i и P_{i+1} для совместного упорядочения содержимого блоков A_i и A_{i+1} может быть осуществлено следующим образом:

- выполнить взаимообмен блоков между процессорами P_i и P_{i+1} ,

- объединить блоки A_i и A_{i+1} на каждом процессоре в один отсортированный блок двойного размера (при исходной упорядоченности блоков A_i и A_{i+1} процедура их объединения сводится к быстрой операции слияния упорядоченных наборов данных),
- разделить полученный двойной блок на две равные части и оставить одну из этих частей (например, с меньшими значениями данных) на процессоре P_i , а другую часть (с большими значениями соответственно) – на процессоре P_{i+1}

Определенная выше операция "сравнить и разделить" может быть использована в качестве *базовой подзадачи* для организации параллельных вычислений, Получаемый в результате параллельныйалгоритм может быть представлен следующим образом:

```
// Parallel odd-even transposition algorithm
ParallelOddEvenSort(double A[], int n) {
  int id = GetProcId();  // Process number
  int np = GetProcNum(); // Number of processes
  for ( int i=0; i<np; i++ ) {
    if ( i%2 == 1 ) { // Odd iteration
      if ( id%2 == 1 ) { // Odd process number
        // Compare-exchange with the right neighbor process
        if ( id < np -1 ) compare_split_min(id+1);
      }
      else
        // Compare-exchange with the left neighbor process
        if ( id > 0 ) compare_ split_max(id-1);
    }
    if ( i%2 == 0 ) { // Even iteration
      if( id%2 == 0 ) { // Even process number
        // Compare-exchange with the right neighbor process
        if ( id < np -1 ) compare_ split_min(id+1);
      }
```

Для пояснения такого параллельного способа сортировки в таблице приведен пример упорядочения данных при $n=16$, $p=4$ (т.е. блок значений на каждом процессоре содержит $n/p=4$ элемента). В первом столбце таблицы приводится номер и тип итерации метода, перечисляются пары процессов, для которых параллельно выполняются операции "сравнить и разделить". Взаимодействующие пары процессов выделены в таблице двойной рамкой. Для каждого шага сортировки показано состояние упорядочиваемого набора данных до и после выполнения итерации.

Таблица Пример сортировки данных параллельным методом чет-нечетной перестановки

№ и тип итерации	Процессоры			
	1	2	3	4
Исходные данные	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
1 нечет (1,2),(3.4)	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
2 чет (2,3)	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75

3 нечет (1,2),(3.4)	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
4 чет (2,3)	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
	2 4 13 14	18 22 29 40	43 43 55 59	71 75 85 88

Масштабирование и распределение подзадач по процессорам

Количество подзадач, соответствует числу имеющихся процессоров и, как результат, необходимости в проведении масштабирования вычислений не возникает. Исходное распределение блоков упорядочиваемого набора данных по процессорам может быть выбрано произвольным образом. Для эффективного выполнения рассмотренного параллельного алгоритма сортировки необходимым является, чтобы процессоры с соседними номерами имели прямые линии связи.

При выполнении **задачи 4** необходимо разработать параллельный алгоритм сортировки.

Программный код параллельного приложения пузырьковой сортировки

ParallelBubbleSort.cpp

```
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <ctime>
#include <cmath>
#include <algorithm>
#include <mpi.h>

#include "ParallelBubbleSort.h"
#include "ParallelBubbleSortTest.h"

using namespace std;

const double RandomDataMultiplier = 1000.0;

int ProcNum = 0;      // Number of available processes
int ProcRank = -1;    // Rank of current process

int main(int argc, char *argv[]) {
  double *pData = 0;
  double *pProcData = 0;
    int DataSize =0;
    int BlockSize=0;

    double *pSerialData = 0;

    double start, finish;
    double duration = 0.0;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

    if(ProcRank == 0)
      printf("Parallel bubble sort program\n");

    // Process initialization
    ProcessInitialization(pData, DataSize, pProcData, BlockSize);
```

```
  if (ProcRank == 0) {
    pSerialData = new double[DataSize];
    CopyData(pData, DataSize, pSerialData);
  }

  start = MPI_Wtime();
  // Distributing the initial data between processes
  DataDistribution(pData, DataSize, pProcData, BlockSize);

  // Testing the distribution
  TestDistribution(pData, DataSize, pProcData, BlockSize);

  // Parallel bubble sort
  ParallelBubble(pProcData, BlockSize);
  // Print the sorted data
  ParallelPrintData(pProcData, BlockSize);

  // Execution of data collection
  DataCollection(pData, DataSize, pProcData,
  BlockSize);TestResult(pData, pSerialData, DataSize);
  finish = MPI_Wtime();

  duration = finish -
  start;if(ProcRank == 0)
    printf("Time of execution: %f\n", duration);

  if (ProcRank == 0)
    delete
    []pSerialData;

  // Process termination
  ProcessTermination(pData, pProcData);

  MPI_Finalize(

  );return 0;
}

// Function for allocating the memory and setting the initial values
void    ProcessInitialization(double    *&pData,    int&    DataSize,    double
*&pProcData, int& BlockSize) {
  setvbuf(stdout, 0, _IONBF,
  0);if(ProcRank == 0) {
    do {
      printf("Enter the size of data to be sorted:
      ");scanf("%d", &DataSize);
      if(DataSize < ProcNum)
        printf("Data size should be greater than number of processes\n");

    } while(DataSize < ProcNum);

    printf("Sorting %d data items\n", DataSize);
  }

  // Broadcasting the data size
  MPI_Bcast(&DataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

  int RestData = DataSize;
  for(int i = 0; i < ProcRank; i++)
    RestData -= RestData / (ProcNum -
    i);
  BlockSize = RestData / (ProcNum - ProcRank);
```

```
  pProcData = new double[BlockSize];

  if(ProcRank == 0) {
    pData = new double[DataSize];

    // Data initalization
    //RandomDataInitialization(pData,
    DataSize);DummyDataInitialization(pData,
    DataSize);
  }
}

// Function for computational process termination
void ProcessTermination(double *pData, double *pProcData)
  {if(ProcRank == 0)
    delete []pData;

  delete []pProcData;
}

// Function for simple setting the data to be sorted
void DummyDataInitialization(double*& pData, int& DataSize)
  {for(int i = 0; i < DataSize; i++)
    pData[i] = DataSize - i;
}

// Function for initializing the data by the random generator
void RandomDataInitialization(double *&pData, int& DataSize) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < DataSize; i++)
    pData[i] = double(rand()) / RAND_MAX * RandomDataMultiplier;
}

// Data distribution among the processes
void DataDistribution(double *pData, int DataSize, double *pProcData,
intBlockSize) {

  // Allocate memory for temporary
  objectsint *pSendInd = new
  int[ProcNum];
  int *pSendNum = new

  int[ProcNum];int RestData =

  DataSize;

  int CurrentSize = DataSize /
  ProcNum;pSendNum[0] = CurrentSize ;
  pSendInd[0] = 0;
  for(int i = 1; i < ProcNum; i++) {



    RestData    -= CurrentSize;
    CurrentSize = RestData / (ProcNum - i);
    pSendNum[i]  = CurrentSize;
    pSendInd[i]  = pSendInd[i - 1] + pSendNum[i - 1];
  }

  MPI_Scatterv(pData, pSendNum, pSendInd, MPI_DOUBLE,
    pProcData,pSendNum[ProcRank], MPI_DOUBLE, 0,
```

```
    MPI_COMM_WORLD);

  // Free the
  memory delete []
  pSendNum;delete
  [] pSendInd;
}

// Function for data collection
void DataCollection(double *pData, int DataSize, double *pProcData,
intBlockSize) {

  // Allocate memory for temporary
  objectsint *pReceiveNum = new
  int[ProcNum];
  int *pReceiveInd = new int[ProcNum];

  int RestData =

  DataSize;

  pReceiveInd[0] = 0;
  pReceiveNum[0] = DataSize /
  ProcNum;for(int i = 1; i <
  ProcNum; i++) {
    RestData -= pReceiveNum[i - 1];
    pReceiveNum[i] = RestData / (ProcNum - i);
    pReceiveInd[i] = pReceiveInd[i - 1] + pReceiveNum[i - 1];
  }

  MPI_Gatherv(pProcData, BlockSize, MPI_DOUBLE, pData,
    pReceiveNum, pReceiveInd, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Free the memory
  delete
  []pReceiveNum;
  delete
  []pReceiveInd;
}

// Parallel bubble sort algorithm
void ParallelBubble(double *pProcData, int BlockSize) {

  // Local sorting the process data
  SerialBubbleSort(pProcData, BlockSize);

  int Offset;
  split_mode
  SplitMode;

  for(int i = 0; i < ProcNum; i++)
    {if((i % 2) == 1) {
      if((ProcRank % 2) == 1) {
        Offset    = 1;
        SplitMode = KeepFirstHalf;
      }
      else {
        Offset    = -1;
        SplitMode = KeepSecondHalf;
      }
    }
    else {
      if((ProcRank % 2) == 1) {
```

```
        Offset  = -1;
        SplitMode = KeepSecondHalf;
      }
      else {
        Offset    = 1;
        SplitMode = KeepFirstHalf;
      }
    }

    // Check the first and last processes
    if((ProcRank == ProcNum - 1) && (Offset == 1))
    continue;if((ProcRank == 0 ) && (Offset == -1))
    continue;

    MPI_Status status;

      int

      DualBlockSize;

    MPI_Sendrecv(&BlockSize, 1, MPI_INT, ProcRank + Offset,
      0,&DualBlockSize, 1, MPI_INT, ProcRank + Offset, 0,
      MPI_COMM_WORLD, &status);

    double *pDualData   = new double[DualBlockSize];
    double *pMergedData = new double[BlockSize + DualBlockSize];

    // Data exchange
    ExchangeData(pProcData,  BlockSize,  ProcRank  +  Offset,
                             pDualData,DualBlockSize);

    // Data merging
    merge(pProcData,  pProcData  +  BlockSize,  pDualData,  pDualData
    +DualBlockSize, pMergedData);

    // Data splitting
    if(SplitMode == KeepFirstHalf)
      copy(pMergedData, pMergedData + BlockSize,
    pProcData);else
      copy(pMergedData   +   BlockSize,   pMergedData
      +                  BlockSize        +DualBlockSize, pProcData);

    delete []pDualData;
    delete []pMergedData;
  }
}

// Function for data exchange between the neighboring processes
void ExchangeData(double *pProcData, int BlockSize, int DualRank,
  double *pDualData, int DualBlockSize) {

  MPI_Status status;
  MPI_Sendrecv(pProcData, BlockSize, MPI_DOUBLE, DualRank,
    0,pDualData, DualBlockSize, MPI_DOUBLE, DualRank, 0,
    MPI_COMM_WORLD, &status);
}

// Function for testing the data distribution
void TestDistribution(double *pData, int DataSize, double *pProcData,
intBlockSize) {
  MPI_Barrier(MPI_COMM_WORLD);
  if (ProcRank == 0) {
    printf("Initial data:\n");
```

```
      PrintData(pData, DataSize);
    }

    MPI_Barrier(MPI_COMM_WORLD);

  for (int i = 0; i < ProcNum; i++) {
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      printf("Block:\n");
      PrintData(pProcData, BlockSize);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

// Function for parallel data output
void ParallelPrintData(double *pProcData, int BlockSize) {
  // Print the sorted data
  for(int i = 0; i < ProcNum; i++) {
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      printf("Proc sorted data: \n");
      PrintData(pProcData, BlockSize);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

// Function for testing the result of parallel bubble sort
void TestResult(double *pData, double *pSerialData, int DataSize) {
  MPI_Barrier(MPI_COMM_WORLD);

  if(ProcRank == 0) {
    SerialBubbleSort(pSerialData, DataSize);
    //SerialStdSort(pSerialData, DataSize);
    if(!CompareData(pData, pSerialData, DataSize))
      printf("The results of serial and parallel algorithms are ""NOT
        identical. Check your code\n");
    else
      printf("The results of serial and parallel algorithms are "
        "identical\n");
  }


}
```

ParallelBubbleSortTest.cpp

```
#include <cstdio>
#include <cstdlib>
#include <algorithm>

#include "ParallelBubbleSortTest.h"

using namespace std;

// Function for copying the sorted data
void CopyData(double *pData, int DataSize, double *pDataCopy) {
  copy(pData, pData + DataSize, pDataCopy);
}

// Function for comparing the data
bool CompareData(double *pData1, double *pData2, int DataSize) {return
  equal(pData1, pData1 + DataSize, pData2);
```

```
}

// Serial bubble sort algorithm
void SerialBubbleSort(double *pData, int DataSize) {
  double Tmp;
  for(int i = 1; i < DataSize; i++)
    for(int j = 0; j < DataSize - i; j++)
      if(pData[j] > pData[j + 1]) {
        Tmp          = pData[j];
        pData[j]     = pData[j + 1];
        pData[j + 1] = Tmp;
      }
}

// Sorting by the standard library algorithm
void SerialStdSort(double *pData, int DataSize) {
  sort(pData, pData + DataSize);
}

// Function for formatted data output
void PrintData(double *pData, int DataSize) {
  for(int i = 0; i < DataSize; i++)
    printf("%7.4f ", pData[i]);
  printf("\n");
}
```

Лабораторная работа №5: «Параллельные алгоритмы обработки графов»

Математические модели в виде *графов* широко используются при моделировании разнообразных явлений, процессов и систем. Как результат, многие теоретические и реальные прикладные задачи могут быть решены при помощи тех или иных процедур анализа графовых моделей. Среди множества этих процедур может быть выделен некоторый определенный набор *типовых алгоритмов обработки графов* . Рассмотрению вопросов теории графов, алгоритмов моделирования, анализу и решению задач на графах посвящено достаточно много различных изданий – см. раздел 11 "Параллельные алгоритмы обработки графов" учебных материалов курса.

Целью данной лабораторной работы является разработка параллельной программы, решающей задачу поиска кратчайших путей используя алгоритм Флойда. Выполнение лабораторной включает выполнение следующих задач:

1. Постановка задачи поиска кратчайших путей.
2. Реализация последовательного алгоритма Флойда.
3. Разработка параллельного алгоритма Флойда.
4. Реализация параллельного алгоритма Флойда.

При выполнении **задачи 1** необходимо изучить последовательный алгоритм Флойда.

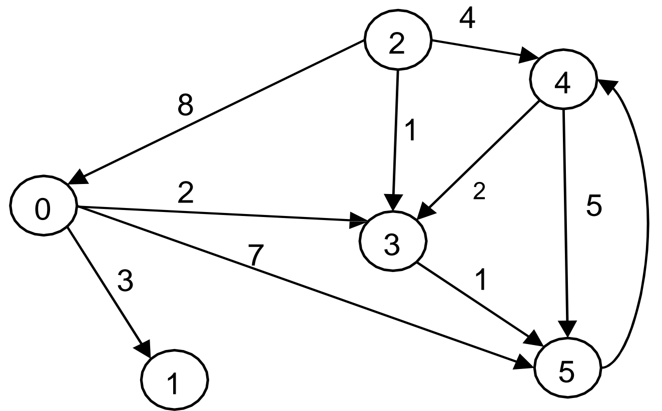
Пусть G есть граф

$G=(V,R)$,

для которого набор вершин v_i , $1 \le i \le n$, задается множеством V, а список дуг графа

$$r_j = (v_{s_j}, v_{t_j}),\ 1 \le j \le m,$$

определяется множеством R. В общем случае дугам графа могут приписываться некоторые числовые характеристики (*веса*) w_j, $1 \le j \le m$ (*взвешенный граф*). Пример взвешенного графа приведен на рисунке



соединены междусобой дугами (т.е. $m \sim n^2$), может быть эффективно обеспечено при помощи *матрицы смежности*

$A = (a_{ij})$, $1 \le i$, $j \le n$,

ненулевые значения элементов которой соответствуют дугам графа

$$a_{ij} = \begin{cases} w(v_i, v_j), & \text{если } (v_i, v_j) \in R, \\ 0, & \text{если } i = j, \\ \infty, & \text{иначе.} \end{cases}$$

(для обозначения отсутствия ребра между вершинами в матрице смежности на соответствующей позиции используется знак бесконечности, при вычислениях знак бесконечности может быть заменен, например, на любое отрицательное число).

$$\begin{pmatrix} 0 & 3 & \infty & 2 & \infty & 7 \\ \infty & 0 & \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & 1 & 4 & \infty \\ \infty & \infty & \infty & 0 & \infty & 1 \\ \infty & \infty & \infty & 2 & 0 & 5 \\ \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Рассмотрим способы параллельной реализации алгоритмов на графах на примере *задачи поиска кратчайших путей* между всеми парами пунктов назначения. Задача состоит в том, что для имеющегося графа *G* требуется найти минимальные длины путей между каждой парой вершин графа. В качестве практического примера можно привести задачу составления маршрута движения транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.

В качестве метода, решающего задачу поиска кратчайших путей между всеми парами пунктов назначения, далее используется *алгоритм Флойда* (*Floyd*) .

Исходной информацией для *задачи поиска кратчайших путей* является взвешенный граф $G = (V, R)$, содержащий n вершин ($|V| = n$), в котором каждому ребру графа приписан неотрицательный вес. Граф будем полагать *ориентированным*, т.е., если из вершины i есть ребро в вершину j, то из этого не следует наличие ребра из j в i. В случае, если вершины все же соединены взаимообратными ребрами, то веса, приписанные им, могут не совпадать. Задача состоит в том, что для имеющегося графа G требуется найти минимальные длины путей между каждой парой вершин графа. В качестве практического примера можно привести задачу составления маршрута движения транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.

Для поиска минимальных расстояний между всеми парами пунктов назначения Флойд предложил алгоритм, сложность которого имеет порядок n^3. В общем виде данный алгоритм может быть представлен следующим образом:

```
// Serial Floyd algorithm
for (k = 0; k < n; k++)

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)

      A[i,j] = min(A[i,j],A[i,k]+A[k,j]);
```

(реализация операции выбора минимального значения *min* должна учитывать способ указания в матрице смежности несуществующих дуг графа). Как можно заметить, в ходе выполнения алгоритма матрица смежности A изменяется, после завершения вычислений в матрице A будет храниться требуемый результат – длины минимальных путей для каждой пары вершин исходного графа.

При выполнении **задачи 2** необходимо реализовать последовательный алгоритм Флойда. Начальный вариант будущей программы представлен в проекте *SerialFloyd*, который содержит некоторую часть исходного кода. В ходе выполнения задачи необходимо дополнить имеющийся вариант программы операциями ввода исходных данных, реализацией алгоритма Флойда и проверкой правильности результатов работы программы.

SerialFloyd.cpp

```
#include <cstdlib>
#include <cstdio>
#include <ctime>
#include <algorithm>

#include "SerialFloyd.h"
#include "SerialFloydTest.h"

using namespace std;


 const double InfinitiesPercent = 50.0;
 const double RandomDataMultiplier = 10;

 int Min(int A, int B) {
   int Result = (A < B) ? A : B;

   if((A < 0) && (B >= 0)) Result = B;
   if((B < 0) && (A >= 0)) Result = A;
   if((A < 0) && (B < 0))  Result = -1;

   return Result;
 }
```

```
int main(int argc, char* argv[]) {
  int *pMatrix;    // Adjacency matrix
  int  Size;       // Size of adjacency matrix

  time_t start, finish;
  double duration = 0.0;

  printf("Serial Floyd algorithm\n");

  // Process initialization
  ProcessInitialization(pMatrix, Size);

  printf("The matrix before Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);

  start = clock();
  // Parallel Floyd algorithm
  SerialFloyd(pMatrix, Size);
  finish = clock();

  printf("The matrix after Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);

  duration = (finish - start) / double(CLOCKS_PER_SEC);

  printf("Time of execution: %f\n", duration);

  // Ending of processing
  ProcessTermination(pMatrix);

  return 0;
}

// Function for allocating the memory and setting the initial values
void ProcessInitiliazation(int *&pMatrix, int& Size) {
  do {
    printf("Enter the number of vertices: ");

    scanf("%d", &Size);

    if(Size <= 0)
      printf("The number of vertices should be greater then zero\n");
  } while(Size <= 0);

  printf("Using graph with %d vertices\n", Size);

  // Allocate memory for the adjacency matrix
  pMatrix = new int[Size * Size];
 // Data initalization
 DummyDataInitialization(pMatrix, Size);
 //RandomDataInitialization(pMatrix, Size);
}

// Function for computational process terminationvoid
ProcessTermination(int *pMatrix) {
 delete []pMatrix;
}

// Function for simple setting the initial data
void DummyDataInitialization(int *pMatrix, int Size) {for(int
 i = 0; i < Size; i++)
   for(int j = i; j < Size; j++) {
     if(i == j) pMatrix[i * Size + j] = 0;
     else
```

```
        if(i == 0) pMatrix[i * Size + j] = j;
        else       pMatrix[i * Size + j] = -1;

      pMatrix[j * Size + i] = pMatrix[i * Size + j];
    }
}

// Function for initializing the data by the random generator
void RandomDataInitialization(int *pMatrix, int Size) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < Size; i++)
    for(int j = 0; j < Size; j++)
      if(i != j) {
        if((rand() % 100) < InfinitiesPercent)
          pMatrix[i * Size + j] = -1;
        else
          pMatrix[i * Size + j] = rand() + 1;
      }
      else
        pMatrix[i * Size + j] = 0;
}

// Serial Floyd algorithm
void SerialFloyd(int *pMatrix, int Size) {
  int t1, t2;
  for(int k = 0; k < Size; k++)
    for(int i = 0; i < Size; i++)
      for(int j = 0; j < Size; j++)
        if((pMatrix[i * Size + k] != -1) &&
           (pMatrix[k * Size + j] != -1)) {
          t1 = pMatrix[i * Size + j];
          t2 = pMatrix[i * Size + k] + pMatrix[k * Size + j];pMatrix[i
          * Size + j] = Min(t1, t2);
      }
}
```

SerialFloydTest.cpp

```
#include <cstdio>
#include "SerialFloydTest.h"

using namespace std;

// Function for formatted matrix output
void PrintMatrix(int *pMatrix, int RowCount, int ColCount) {
  for(int i = 0; i < RowCount; i++) {
    for(int j = 0; j < ColCount; j++)
      printf("%7d", pMatrix[i * ColCount + j]);
    printf("\n");
  }
}
```

При выполнении **задачи 3** необходимо изучить принципы распараллеливания алгоритма Флойда. Для этого необходимо провести декомпозицию задачи, выделить информационные взаимодействия между подзадачами, определить вычислительную схему и выполнить распределение набора подзадач по вычислительным устройствам.

Определение подзадач

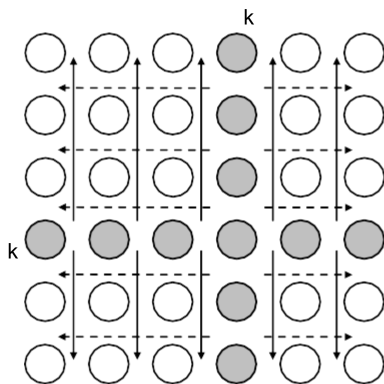
Как следует из общей схемы алгоритма Флойда, основная вычислительная нагрузка при решении задачи поиска кратчайших путей состоит в выполнении операции выбора

минимальных значения. Данная операция является достаточно простой и ее распараллеливание не приведет к заметному ускорению вычислений. Более эффективный способ организации параллельных вычислений может состоять в одновременном выполнении нескольких операций обновления значений матрицы A .

Как результат, необходимые условия для организации параллельных вычислений обеспечены и, тем самым, в качестве *базовой подзадачи* может быть использована операция обновления элементов матрицы A (для указания подзадач будем использовать индексы обновляемых в подзадачах элементов).

Выделение информационных зависимостей

Выполнение вычислений в подзадачах становится возможным только тогда, когда каждая подзадача (i,j) содержит необходимые для расчетов элементы A_{ij} , A_{ik} , A_{kj} матрицы A . Для исключения дублирования данных разместим в подзадаче (i,j) единственный элемент A_{ij} , тогда получение всех остальных необходимых значений может быть обеспечено только при помощи передачи данных. Таким образом, каждый элемент A_{kj} строки k матрицы A должен быть передан всем подзадачам (k,j), $1 \le j \le n$,
а каждый элемент A_{ik} столбца k матрицы A должен быть передан всем подзадачам (i,k), $1 \le i \le n$,



Информационная зависимость базовых подзадач (стрелками показаны направления обмена значениями на итерации k)

Масштабирование и распределение подзадач по процессорам

Как правило, число доступных процессоров p существенно меньше, чем число базовых задач n^2 ($p << n^2$). Возможный способ укрупнения вычислений состоит в использовании *ленточной схемы* разбиения матрицы A – такой подход соответствует объединению в рамках одной базовой подзадачи вычислений, связанных с обновлением элементов одной или нескольких строк (*горизонтальное* разбиение) или столбцов (*вертикальное* разбиение) матрицы A . Эти два типа разбиения практически равноправны – учитывая дополнительный момент, что для алгоритмического языка C массивы располагаются по строкам, будем рассматривать далее только разбиение матрицы A на горизонтальные полосы.

Следует отметить, при таком способе разбиения данных на каждой итерации алгоритма Флойда потребуется передавать между подзадачами только элементы одной из строк матрицы A . Для эффективного выполнения подобной коммуникационной операции топология сети должна представлять собой гиперкуб или полный граф.

При выполнении **задачи 4** необходимо разработать параллельный алгоритм Флойда.

ParallelFloyd.cpp

```
#include <cstdlib> #include <cstdio> #include <ctime> #include
<algorithm>#include <mpi.h>

#include "ParallelFloyd.h" #include "ParallelFloydTest.h"

using namespace std;
int ProcRank;
int ProcNum;
const double InfinitiesPercent = 50.0; const double
RandomDataMultiplier = 10;

int Min(int A, int B) {
  int Result = (A < B) ? A : B;

  if((A < 0) && (B >= 0)) Result = B; if((B < 0) && (A >= 0))
  Result = A; if((A < 0) && (B < 0))  Result = -1;

  return Result;
}

int main(int argc, char* argv[]) {
  int *pMatrix;       // Adjacency matrix
  int  Size;          // Size of adjacency matrixint *pProcRows;   //
  Process rows
  int  RowNum;        // Number of process rows

  double start, finish; double duration =
  0.0;

  int *pSerialMatrix = 0;MPI_Init(&argc,

  &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum); MPI_Comm_rank(MPI_COMM_WORLD,
  &ProcRank);

  if(ProcRank == 0)
    printf("Parallel Floyd algorithm\n");

  // Process initialization
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);

  if (ProcRank == 0) {
      // Matrix copying
    pSerialMatrix = new int[Size * Size]; CopyMatrix(pMatrix, Size,
    pSerialMatrix);
    }

   start = MPI_Wtime();
   // Distributing the initial data between processes
   DataDistribution(pMatrix, pProcRows, Size, RowNum);
   // Testing the distribution
   //TestDistribution(pMatrix, pProcRows, Size, RowNum);

   // Parallel Floyd algorithm
   ParallelFloyd(pProcRows, Size, RowNum);
   //ParallelPrintMatrix(pProcRows, Size, RowNum);

   // Process data collection
   ResultCollection(pMatrix, pProcRows, Size, RowNum);
   //if(ProcRank == 0)
```

```
  //  PrintMatrix(pMatrix, Size, Size);
  finish = MPI_Wtime();

  //TestResult(pMatrix, pSerialMatrix, Size);

  duration = finish - start;
  if(ProcRank == 0)
    printf("Time of execution: %f\n", duration);

  if (ProcRank == 0)
    delete []pSerialMatrix;

  // Process termination
  ProcessTermination(pMatrix, pProcRows);

  MPI_Finalize();
  return 0;
}

// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int *&pProcRows, int& Size, int&
RowNum) {
  setvbuf(stdout, 0, _IONBF, 0);

  if(ProcRank == 0) {
    do {
      printf("Enter the number of vertices: ");

      scanf("%d", &Size);

      if(Size < ProcNum)
        printf("The number of vertices should be greater then number of
processes\n");
    } while(Size < ProcNum);

    printf("Using graph with %d vertices\n", Size);
  }

  // Broadcast the number of vertices
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Number of rows for each process
  int RestRows = Size;
  for(int i = 0; i < ProcRank; i++)
    RestRows = RestRows - RestRows / (ProcNum - i);
  RowNum = RestRows / (ProcNum - ProcRank);


// Allocate memory for the current process rows

pProcRows = new int[Size * RowNum];
  if(ProcRank == 0) {
    // Allocate memory for the adjacency matrix
    pMatrix = new int[Size * Size];

    // Data initalization
    DummyDataInitialization(pMatrix, Size);
    //RandomDataInitialization(pMatrix, Size);
  }
}

// Function for computational process termination
void ProcessTermination(int *pMatrix, int *pProcRows) {
```

```
  if(ProcRank == 0)
    delete []pMatrix;

  delete []pProcRows;
}

// Function for simple setting the initial data
void DummyDataInitialization(int *pMatrix, int Size) {
  for(int i = 0; i < Size; i++)
    for(int j = i; j < Size; j++) {
      if(i == j) pMatrix[i * Size + j] = 0;
      else
        if(i == 0) pMatrix[i * Size + j] = j;
        else       pMatrix[i * Size + j] = -1;

      pMatrix[j * Size + i] = pMatrix[i * Size + j];
    }
}

// Function for setting the data by the random generator
void RandomDataInitialization(int *pMatrix, int Size) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < Size; i++)
    for(int j = 0; j < Size; j++)
      if(i != j) {
        if((rand() % 100) < InfinitiesPercent)
          pMatrix[i * Size + j] = -1;
        else
          pMatrix[i * Size + j] = rand() + 1;
      }
      else
        pMatrix[i * Size + j] = 0;
}

// Data distribution among the processes
void DataDistribution(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  int *pSendNum; // The number of elements sent to the process
  int *pSendInd; // The index of the first data element sent to the process

  int RestRows = Size; // Number of rows, that havenTt been distributed yet

  // Allocate memory for temporary objects
  pSendInd = new int[ProcNum];
  pSendNum = new int[ProcNum];

  // Define the disposition of the matrix rows for current process
  RowNum = Size / ProcNum;
  pSendNum[0] = RowNum * Size;


  pSendInd[0] = 0;
  for (int i = 1; i < ProcNum; i++) {
    RestRows   -= RowNum;
    RowNum      = RestRows / (ProcNum - i);
    pSendNum[i] = RowNum * Size;
    pSendInd[i] = pSendInd[i - 1] + pSendNum[i - 1];
  }

  // Scatter the rows
  MPI_Scatterv(pMatrix, pSendNum, pSendInd, MPI_INT,
    pProcRows, pSendNum[ProcRank], MPI_INT, 0, MPI_COMM_WORLD);
```

```
  // Free allocated memory
  delete []pSendNum;
  delete []pSendInd;
}

// Function for process result collection
void ResultCollection(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  int *pReceiveNum;  // Number of elements, that current process sends
  int *pReceiveInd;  /* Index of the first element from current process
                        in result vector */
  int RestRows = Size; // Number of rows, that havenTt been gathered yet

  // Allocate memory for temporary objects
  pReceiveNum = new int[ProcNum];
  pReceiveInd = new int[ProcNum];

  // Define the disposition of the result vector block of current process
  RowNum = Size / ProcNum;
  pReceiveInd[0] = 0;
  pReceiveNum[0] = RowNum * Size;

  for(int i = 1; i < ProcNum; i++) {
    RestRows      -= RowNum;
    RowNum = RestRows / (ProcNum - i);
    pReceiveNum[i] = RowNum * Size;
    pReceiveInd[i] = pReceiveInd[i - 1] + pReceiveNum[i - 1];
  }

  // Gather the whole matrix on process with rank 0
  MPI_Gatherv(pProcRows, pReceiveNum[ProcRank], MPI_INT,
    pMatrix, pReceiveNum, pReceiveInd, MPI_INT, 0, MPI_COMM_WORLD);

  // Free allocated memory
  delete []pReceiveNum;
  delete []pReceiveInd;
}

// Parallel Floyd algorithm
void ParallelFloyd(int *pProcRows, int Size, int RowNum) {
  int *pRow = new int[Size];
  int t1, t2;
  for(int k = 0; k < Size; k++) {
    // Distribute row among all processes
    RowDistribution(pProcRows, Size, RowNum, k, pRow);

    // Update adjacency matrix elements
    for(int i = 0; i < RowNum; i++)
      for(int j = 0; j < Size; j++)
        if( (pProcRows[i * Size + k] != -1) &&
            (pRow     [j]            != -1)) {
          t1 = pProcRows[i * Size + j];

          t2 = pProcRows[i * Size + k] + pRow[j];

        pProcRows[i * Size + j] = Min(t1, t2);
      }
  }

  delete []pRow;
}
```

```
// Function for row broadcasting among all processes
void RowDistribution(int *pProcRows, int Size, int RowNum, int k, int
*pRow) {
  int ProcRowRank;  // Process rank with the row k
  int ProcRowNum;   // Process row number

  // Finding the process rank with the row k
  int RestRows = Size;
  int Ind = 0;
  int Num = Size / ProcNum;

  for(ProcRowRank = 1; ProcRowRank < ProcNum + 1; ProcRowRank ++) {
    if(k < Ind + Num ) break;
    RestRows -= Num;
    Ind      += Num;
    Num       = RestRows / (ProcNum - ProcRowRank);
  }
  ProcRowRank = ProcRowRank - 1;
  ProcRowNum  = k - Ind;

  if(ProcRowRank == ProcRank)
    // Copy the row to pRow array
    copy(&pProcRows[ProcRowNum*Size],&pProcRows[(ProcRowNum+1)*Size],pRow);

  // Broadcast row to all processes
  MPI_Bcast(pRow, Size, MPI_INT, ProcRowRank, MPI_COMM_WORLD);
}

// Function for formatted output of all stripes
void ParallelPrintMatrix(int *pProcRows, int Size, int RowNum) {
  for(int i = 0; i < ProcNum; i++) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      fflush(stdout);
      printf("Proc rows:\n");
      fflush(stdout);
      PrintMatrix(pProcRows, RowNum, Size);
      fflush(stdout);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

// Function for testing the data distribution
void TestDistribution(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  MPI_Barrier(MPI_COMM_WORLD);
  if (ProcRank == 0) {
    printf("Initial adjacency matrix:\n");
    PrintMatrix(pMatrix, Size, Size);
  }

  MPI_Barrier(MPI_COMM_WORLD);
  ParallelPrintMatrix(pProcRows, Size, RowNum);
}

// Testing the result of parallel Floyd algorithm
void TestResult(int *pMatrix, int *pSerialMatrix, int Size) {
  MPI_Barrier(MPI_COMM_WORLD);

  if(ProcRank == 0) {
    SerialFloyd(pSerialMatrix, Size);
    if(!CompareMatrices(pMatrix, pSerialMatrix, Size)) {
```

```
      printf("Results of serial and parallel algorithms are "
        "NOT identical. Check your code\n");
    }
    else {
      printf("Results of serial and parallel algorithms are "
        "identical\n");
    }
  }
}
```

ParallelFloydTest.cpp

```
#include <cstdio>
#include <algorithm>
using namespace std;

#include "ParallelFloyd.h"
#include "ParallelFloydTest.h"

// Function for copying the matrix
void CopyMatrix(int *pMatrix, int Size, int *pMatrixCopy) {
  copy(pMatrix, pMatrix + Size * Size, pMatrixCopy);
}

// Function for comparing the matrices
bool CompareMatrices(int *pMatrix1, int *pMatrix2, int Size) {return
  equal(pMatrix1, pMatrix1 + Size * Size, pMatrix2);
}

// Serial Floyd algorithm
void SerialFloyd(int *pMatrix, int Size) {
  int t1, t2;
  for(int k = 0; k < Size; k++)
    for(int i = 0; i < Size; i++)
      for(int j = 0; j < Size; j++)
        if((pMatrix[i * Size + k] != -1) &&
           (pMatrix[k * Size + j] != -1)) {
          t1 = pMatrix[i * Size + j];
          t2 = pMatrix[i * Size + k] + pMatrix[k * Size + j];pMatrix[i
          * Size + j] = Min(t1, t2);
      }
}

// Function for formatted matrix output
void PrintMatrix(int *pMatrix, int RowCount, int ColCount) {
  for(int i = 0; i < RowCount; i++) {
    for(int j = 0; j < ColCount; j++) {
      printf("%7d", pMatrix[i * ColCount + j]);
      fflush(stdout);
    }
    printf("\n");
    fflush(stdout);}
```

Лабораторная работа №6: «Параллельные алгоритмы решения дифференциальных уравнений в частных производных»

Дифференциальные уравнения в частных производных представляют собой широко применяемый математический аппарат при разработке моделей в самых разных областях науки и техники. К сожалению, явное решение этих уравнений в аналитическом виде оказывается возможным только в частных простых случаях, и, как результат, возможность анализа математических моделей, построенных на основе дифференциальных уравнений, обеспечивается при помощи приближенных численных методов решения. Объем выполняемых при этом вычислений обычно является значительным, и использование высокопроизводительных вычислительных систем является традиционным для данной области вычислительной математики. Проблематика численного решения дифференциальных уравнений в частных производных является областью интенсивных исследований.

Целью данной лабораторной работы является разработка параллельной программы, которая обеспечивает решение одной из задач, описываемой дифференциальным уравнением в частных производных – *задачи Дирихле для уравнения Пуассона*. Выполнение лабораторной включает выполнение следующих задач:

1. Постановка задачи Дирихле
2. Реализация последовательного алгоритма Гаусса – Зейделя решения задачи Дирихле
3. Разработка параллельного алгоритма Гаусса – Зейделя решения задачи Дирихле
4. Реализация параллельного алгоритма Гаусса – Зейделя решения задачи Дирихле

Псевдокод для алгоритма Гаусса – Зейделя решения задачи Дирихле может быть представлен следующим образом:

```
// Serial Gauss-Seidel algorithm
do {
  dmax = 0; // maximal variation of the values u
  for ( i=1; i<N+1; i++ )
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                u[i][j-1]+u[i][j+1]–h*h*f[i][j]);
      dm = fabs(temp-u[i][j]);
      if ( dmax < dm ) dmax = dm;
    }
} while ( dmax > eps );
```

При выполнении **задачи 2** необходимо реализовать последовательный алгоритм Гаусса – Зейделя решения задачи Дирихле. Начальный вариант будущей программы представлен в проекте *SerailGaussSeidel*, который содержит часть исходного кода и в котором заданы необходимые параметры проекта. В ходе выполнения задания необходимо дополнить имеющийся вариант программы операциями ввода исходных данных, решения задачи Дирихле с использованием алгоритма Гаусса- Зейделя и вывода результатов.

SerailGaussSeidel.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>


// Function for the Gauss-Seidel algoritm
void ResultCalculation(double* pMatrix, int Size, double &Eps,int
  &Iterations) {
  double dm, dmax,temp;
  int i, j; // Loop variables
  Iterations = 0;
  do {
    dmax = 0;
    for (i = 1; i < Size - 1; i++)
      for(j = 1; j < Size - 1; j++) {
        temp = pMatrix[Size * i + j];
        pMatrix[Size * i + j] = 0.25 * (pMatrix[Size * i + j + 1] +
                                         pMatrix[Size * i + j - 1] +
                                         pMatrix[Size * (i + 1) + j] +
                                         pMatrix[Size * (i - 1) + j]);
        dm = fabs(pMatrix[Size * i + j] - temp);if
        (dmax < dm) dmax = dm;
       }
       Iterations++;
   }
   while (dmax > Eps);
 }

 // Function for computational process termination
 void ProcessTermination(double* pMatrix) {
   delete [] pMatrix;
 }
 // Function for formatted matrix output
 void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
   int i, j; // Loop variables
   for (i=0; i<RowCount; i++) {
     for (j=0; j<ColCount; j++)
       printf("%7.4f ", pMatrix[i*ColCount+j]);
     printf("\n");
   }
 }
 // Function for simple setting the grid node values
 void DummyDataInitialization (double* pMatrix, int Size) {
   int i, j;  // Loop variables
```

```
  double h = 1.0 / (Size - 1);
  // Setting the grid node values
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
      if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))
        pMatrix[i*Size+j] = 100;
      else
        pMatrix[i*Size+j] = 0;
  }
}
// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, int &Size, double &Eps) {
  // Setting the grid size
  do {
    printf("\nEnter the grid size: ");
    scanf("%d", &Size);
    printf("\nChosen grid size = %d", Size);
    if (Size <= 2)
      printf("\nSize of grid must be greater than 2!\n");
  } while (Size <= 2);
  // Setting the required accuracy
  do {
    printf("\nEnter the required accuracy: ");
    scanf("%lf", &Eps);
    printf("\nChosen accuracy = %lf", Eps);
    if (Eps <= 2)
      printf("\nAccuracy must be greater than 0!\n");
  }   while (Eps <= 0);

  // Memory allocation
  pMatrix = new double [Size*Size];

  // Setting the grid node values
  DummyDataInitialization(pMatrix, Size);
}

void main() {
  double* pMatrix;    // Matrix of the grid nodes
  int     Size;       // Matrix size
  double  Eps;        // Required accuracy
  int     Iterations; // Iteration number
 printf ("Serial Gauss - Seidel algorithm\n");


 // Process initialization
 ProcessInitialization(pMatrix, Size, Eps);

 // Matrix output

 printf ("Initial Matrix: \n");
 PrintMatrix (pMatrix, Size, Size);


 // The Gauss-Seidel method ResultCalculation(pMatrix,
 Size, Eps, Iterations);


 // Printing the result

 printf("\n Number of iterations: %d\n",Iterations);
 printf ("\n Result matrix: \n");

 PrintMatrix (pMatrix, Size, Size);
 getch();
```

```
  // Computational process termination
  ProcessTermination(pMatrix);
}
```

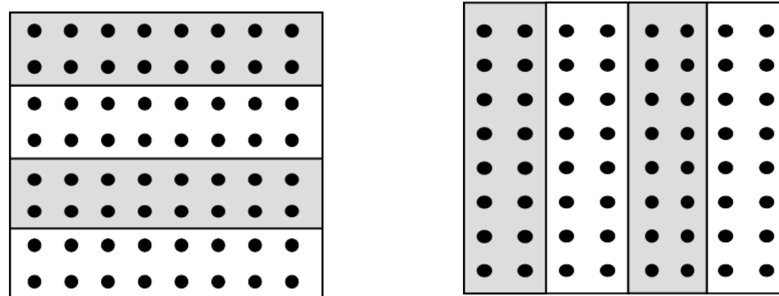
При разработке параллельных алгоритмов необходимо выбрать способ разделения обрабатываемых данных между вычислительными серверами. При построении параллельных способов решения задачи Дирихле возможны два различных способа разделения данных – *одномерная* или *ленточная* схема или *двухмерное* или *блочное* разбиение вычислительной сетки.

Определение подзадач

При ленточном разбиении область расчетов делится на горизонтальные или вертикальные полосы. Число полос определяется количеством параллельных процессов, размер полос обычно является одинаковым, узлы горизонтальных границ (первая и последняя строки) включаются в первую и последнюю полосы соответственно. Полосы для обработки распределяются между процессами.

Разделение строк и столбцов на полосы в большинстве случаев происходит на *непрерывной* (*последовательной*) основе и именно такой подход используется в данной лабораторной работе.

Основной момент при организации вычислений с подобным разделением данных состоит в том, что на процессор, выполняющий обработку какой-либо полосы, должны быть продублированы граничные строки предшествующей и следующей полос вычислительной сетки. Продублированные граничные строки полос используются только при проведении расчетов, пересчет же этих строк происходит в полосах своего исходного месторасположения. Тем самым дублирование граничных строк должно осуществляться перед началом выполнения каждой очередной итерации метода сеток.



Способы распределения элементов матрицы между процессорами вычислительной системы

В качестве начального варианта рассмотрим предельный случай, когда количество процессоров совпадает с числом внутренних строк сетки, т.е. $p=N$. В такой ситуации полоса каждого процессора состоит из трех строк, из которых только одна является перевычисляемой, а две других строки дублируются с соседних процессов. Примем далее все вычисления, связанные с обработкой каждой из таких полос, в качестве *базовой вычислительной подзадачи*.

Выделение информационных зависимостей

Параллельный вариант метода сеток при ленточном разделении данных состоит в обработке полос на всех имеющихся процессорах одновременно в соответствии со следующей схемой работы:

```
// Схема Гаусса-Зейделя, ленточное разделение данных
// действия, выполняемые на каждом процессоре
do {
  // <обмен граничных строк полос с соседями>
  // <обработка полосы>
  // <вычисление общей погрешности вычислений dmax>}
 while ( dmax > eps ); // eps - точность решения
```

Для конкретизации представленных в алгоритме действий введем обозначения:

- **ProcNum** – номер процессора, на котором выполняются описываемые действия,
- **PrevProc, NextProc** – номера соседних процессоров, содержащих предшествующую и следующую полосы,
- **NP** – количество процессоров,
- **M** – количество строк в полосе (без учета продублированных граничных строк),
- **N** – количество внутренних узлов в строке сетки (т.е. всего в строке N+2 узла).

Для нумерации строк полосы будем использовать нумерацию, при которой строки 0 и M+1 есть продублированные из соседних полос граничные строки, а строки собственной полосы процессора имеют номера от 1 до M.

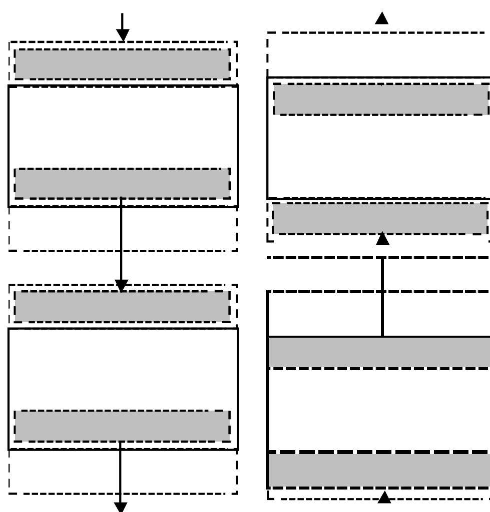


Схема передачи граничных строк между соседними процессорами

Процедура обмена граничных строк между соседними процессорами может быть разделена на две последовательные операции, во время первой из которых каждый процессор передает свою нижнюю граничную строку следующему процессору и принимает такую же строку от предыдущего процессора (см. рис.). Вторая часть передачи строк выполняется в обратном направлении: процессоры передают свои верхние граничные строки своим предыдущим соседям и принимают переданные строки от следующих процессоров.

Выполнение подобных операций передачи данных в общем виде может быть представлено следующим образом (для операций передачи данных используется псевдокод, близкий к функциям MPI):

```
// передача нижней граничной строки следующему процессору
// и прием передаваемой строки от предыдущего процессора
Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
```

Реализация подобной объединенной функции *Sendrecv* обычно осуществляется таким образом, чтобы обеспечить и корректную работу на крайних процессорах, когда не нужно выполнять одну из операций передачи или приема, и организацию чередования процедур передачи на процессорах для ухода от тупиковых ситуаций, и возможности параллельного выполнения всех необходимых пересылок данных.

Для вычисления общей для всех процессоров погрешности вычислений может быть использована каскадная схема, для выполнения которой в MPI имеется функция *MPI_Allreduce*.

Общая схема вычислений на каждом процессоре может быть представлена на псевдокоде в следующем виде**:**

```
// Схема Гаусса-Зейделя, ленточное разделение данных
// действия, выполняемые на каждом процессоре
do {
  // обмен граничных строк полос с соседями
Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
Sendrecv(u[1][*],N+2,PrevProc,u[M+1][*],N+2,NextProc);
  // <обработка полосы с оценкой погрешности dm>
  // вычисление общей погрешности вычислений dmax
  Allreduce(dm,dmax,MAX,0);
} while ( dmax > eps ); // eps - точность решения
```

Масштабирование и распределение подзадач по процессорам

Как правило, число доступных процессоров p существенно меньше, чем число базовых подзадач N ($p << N$). Возможный способ укрупнения вычислений состоит в использовании *ленточной схемы* разбиения матрицы A – такой подход соответствует объединению в рамках одной базовой подзадачи вычислений, связанных с обновлением элементов одной или нескольких строк (*горизонтальное* разбиение) или столбцов (*вертикальное* разбиение) матрицы A . Эти два типа разбиения практически равноправны – учитывая дополнительный момент, что для алгоритмического языка С массивы располагаются по строкам, будем рассматривать далее только разбиение матрицы A на горизонтальные полосы.

При выполнении **задачи 4** необходимо разработать параллельный алгоритм Гаусса – Зейделя решения задачи Дирихле.

Программный код параллельного приложения для алгоритма Гаусса-Зейделя

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>
#include <mpi.h>
#include <algorithm.h>
```

```
static int ProcNum = 0;    // Number of available processes
static int ProcRank = -1; // Rank of current process

// Function for distribution of the grid rows among the processes
void DataDistribution(double* pMatrix, double* pProcRows, int RowNum,
  int Size) {
  int *pSendNum; // Number of elements sent to the process
  int *pSendInd; // Index of the first data element sent to the process
  int RestRows=Size;
  // Alloc memory for temporary objects
  pSendInd = new int [ProcNum];
  pSendNum = new int [ProcNum];
  // Define the disposition of the matrix rows for current process
  RowNum = (Size-2)/ProcNum+2;
  pSendNum[0] = RowNum*Size;
  pSendInd[0] = 0;
  for (int i=1; i<ProcNum; i++) {
    RestRows = RestRows - RowNum + 2;
    RowNum = (RestRows-2)/(ProcNum-i)+2;
    pSendNum[i] = (RowNum)*Size;
    pSendInd[i] = pSendInd[i-1]+pSendNum[i-1]-Size;
  }
  // Scatter the rows
  MPI_Scatterv(pMatrix , pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);
  delete []pSendInd;
  delete []pSendNum;
}
// Function for computational process termination
void ProcessTermination (double* pMatrix, double* pProcRows) {
  if (ProcRank == 0)
    delete [] pMatrix;
  delete [] pProcRows;
}

// Function for formatted matrix output
void PrintMatrix(double *pMatrix, int RowCount, int ColCount){
  int i,j; // Loop variables
  for(int i=0; i < RowCount; i++) {
    for(j=0; j < ColCount; j++)
      printf("%7.4f ", pMatrix[i*ColCount+j]);
    printf("\n");
  }
}
// Function for the execution of the Gauss-Seidel method iteration
double IterationCalculation(double* pProcRows, int Size, int RowNum) {
  int i, j;  // Loop variables
  double dm, dmax,temp;
  dmax = 0;
  for (i = 1; i < RowNum-1; i++)
    for(j = 1; j < Size-1; j++) {
      temp = pProcRows[Size * i + j];
      pProcRows[Size * i + j] = 0.25 * (pProcRows[Size * i + j + 1] +
                                        pProcRows[Size * i + j - 1] +
                                        pProcRows[Size * (i + 1) + j] +
                                        pProcRows[Size * (i - 1) + j]);
      dm = fabs(pProcRows[Size * i + j] - temp);
      if (dmax < dm) dmax = dm;
  }
  return dmax;
}
```

```
// Function for testing the data distribution
void TestDistribution(double* pMatrix, double* pProcRows, int Size,
  int RowNum) {
  if (ProcRank == 0) {
    printf("Initial Matrix: \n");
    PrintMatrix(pMatrix, Size, Size);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  for (int i=0; i<ProcNum; i++) {
    if (ProcRank == i) {
      printf("\nProcRank = %d \n", ProcRank);
//      fprintf(" Matrix Stripe:\n");
      PrintMatrix(pProcRows, RowNum, Size);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

// Function for simple setting the grid node values
void DummyDataInitialization (double* pMatrix, int Size) {
  int i, j;  // Loop variables
  double h = 1.0 / (Size - 1);
  // Setting the grid node values
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
      if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))
        pMatrix[i*Size+j] = 100;
      else
        pMatrix[i*Size+j] = 0;
  }
}

// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, double* &pProcRows,int &Size,
  int &RowNum, double &Eps) {
  int RestRows; // Number of rows, that haven't been distributed yet
  // Setting the grid size
  if (ProcRank == 0) {
    do {
      printf("\nEnter the grid size: ");
      scanf("%d", &Size);
      if (Size <= 2) {
         printf("\n Size of grid must be greater than 2! \n");
      }
      if (Size < ProcNum) {
        printf("Size of grid must be greater than"
               "the number of processes! \n ");
      }
    }
    while ( (Size <= 2) || (Size < ProcNum));

    // Setting the required accuracy
    do {
      printf("\nEnter the required accuracy: ");
      scanf("%lf", &Eps);
      printf("\nChosen accuracy = %lf", Eps);
      if (Eps <= 0)
        printf("\nAccuracy must be greater than 0!\n");
    }
    while (Eps <= 0);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&Eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
  // Define the number of matrix rows stored on each process
  RestRows = Size;
  for (i=0; i<ProcRank; i++)
    RestRows = RestRows-RestRows/(ProcNum-i);
  RowNum = (RestRows-2)/(ProcNum-ProcRank)+2

  // Memory allocation
  pProcRows = new double [RowNum*Size];
  // Define the values of initial objects' elements
  if (ProcRank == 0) {
    // Initial matrix exists only on the pivot process
    pMatrix = new double [Size*Size];
    // Values of elements are defined only on the pivot process
    DummyDataInitialization(pMatrix, Size);
  }
}
// Function for exchanging the boundary rows of the process stripes
void ExchangeData(double* pProcRows, int Size, int RowNum) {
  MPI_Status status;
  int NextProcNum = (ProcRank == ProcNum-1)? MPI_PROC_NULL : ProcRank + 1;
  int PrevProcNum    = (ProcRank == 0)? MPI_PROC_NULL : ProcRank - 1;
  // Send to NextProcNum and receive from PrevProcNum
  MPI_Sendrecv(pProcRows + Size * (RowNum - 2),Size, MPI_DOUBLE,
    NextProcNum, 4, pProcRows, Size, MPI_DOUBLE, PrevProcNum, 4,
    MPI_COMM_WORLD, &status);
  // Send to PrevProcNum and receive from NextProcNum
  MPI_Sendrecv(pProcRows + Size, Size, MPI_DOUBLE, PrevProcNum, 5,
    pProcRows + (RowNum - 1) * Size, Size,MPI_DOUBLE, NextProcNum, 5,
    MPI_COMM_WORLD, &status);
}

// Function for the parallel Gauss - Seidel method
void ParallelResultCalculation (double *pProcRows, int Size, int RowNum,
  double Eps, int &Iterations) {
  double ProcDelta,Delta;
  Iterations=0;
  do {
    Iterations++;
    // Exchanging the boundary rows of the process stripe
    ExchangeData(pProcRows, Size,RowNum);

    // The Gauss-Seidel method iteration
    ProcDelta = IterationCalculation(pProcRows, Size, RowNum);

    // Calculating the maximum value of the deviation
    MPI_Allreduce(&ProcDelta, &Delta, 1,MPI_DOUBLE, MPI_MAX,
      MPI_COMM_WORLD);
  } while ( Delta > Eps);
}
// Function for gathering the result vector
void ResultCollection(double *pMatrix, double* pProcResult, int Size,
  int RowNum) {
  int *pReceiveNum; // Number of elements, that current process sends
  int *pReceiveInd;  // Index of the first element of the received block
  int RestRows = Size;
  int i;             // Loop variable

  // Alloc memory for temporary objects
  pReceiveNum = new int [ProcNum];
  pReceiveInd = new int [ProcNum];

  // Define the disposition of the result vector block of current processor
```

```
  pReceiveInd[0] = 0;
  RowNum = (Size-2)/ProcNum+2;
  pReceiveNum[0] = RowNum*Size;
  for ( i=1; i < ProcNum; i++){
    RestRows = RestRows - RowNum + 1;
    RowNum = (RestRows-2)/(ProcNum-i)+2;
    pReceiveNum[i] = RowNum*Size;
    pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1]-Size;
  }

  // Gather the whole result vector on every processor
  MPI_Allgatherv(pProcRows, pReceiveNum[ProcRank], MPI_DOUBLE, pMatrix,
    pReceiveNum, pReceiveInd, MPI_DOUBLE, MPI_COMM_WORLD);

  // Free the memory
  delete [] pReceiveNum;
  delete [] pReceiveInd;
}

// Function for the serial Gauss - Seidel method
void SerialResultCalculation(double *pMatrixCopy, int Size, double Eps,
  int &Iter){
  int i, j;  // Loop variables
  double dm, dmax,temp;
  Iter = 0;
  do {
    dmax = 0;
    for (i = 1; i < Size - 1; i++)
      for(j = 1; j < Size - 1; j++) {
        temp = pMatrixCopy[Size * i + j];
        pMatrixCopy[Size * i + j] = 0.25 * (pMatrixCopy[Size * i + j + 1] +
                                    pMatrixCopy[Size * i + j - 1] +
                                    pMatrixCopy[Size * (i + 1) + j] +
                                    pMatrixCopy[Size * (i - 1) + j]);
        dm = fabs(pMatrixCopy[Size * i + j] - temp);
        if (dmax < dm) dmax = dm;
      }
      Iter++;
  }
  while (dmax > Eps);

}
// Function to copy the initial data
void CopyData(double *pMatrix, int Size, double *pSerialMatrix) {
  copy(pMatrix, pMatrix + Size, pSerialMatrix);
}

// Function for testing the computation result
void TestResult(double* pMatrix, double* pSerialMatrix, int Size,
  double Eps) {
  int equal = 0; // =1, if the matrices are not equal
  int Iter;

  if (ProcRank == 0) {
    SerialResultCalculation(pSerialMatrix, Size, Eps, Iter);
    for (int i=0; i<Size*Size; i++) {
      if (fabs(pSerialMatrix[i]-pMatrix[i]) >= Eps)
        equal = 1;break;
    }
    if (equal == 1)
      printf("The results of serial and parallel algorithms"
             "are NOT identical. Check your code.");
    else
```

```
      printf("The results of serial and parallel algorithms"
             "are identical.");
    }
}

// Function for setting the grid node values by a random generator
void RandowmDataInitialization (double* pMatrix, int Size) {
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  // Setting the grid node values
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
      if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))
        pMatrix[i*Size+j] = 100;
      else
        pMatrix[i*Size+j] = rand()/double(1000);
  }
}

void main(int argc, char* argv[]) {
  double* pMatrix;       // Matrix of the grid nodes
  double* pProcRows;     // Stripe of the matrix on current process
  double* pSerialMatrix; // Result of the serial method
  int     Size;          // Matrix size
  int     RowNum;        // Number of rows in matrix stripe
  double  Eps;           // Required accuracy
  int     Iterations;    // Iteration number
  double currDelta, delta;

  setvbuf(stdout, 0, _IONBF, 0);
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  if(ProcRank == 0) {
    printf("Parallel Gauss - Seidel algorithm \n");
    fflush(stdout);
  }
  // Process initialization
  ProcessInitialization (pMatrix,pProcRows,Size,RowNum,Eps);

  // Creating the copy of the initial data
  if (ProcRank == 0) {
    pSerialMatrix = new double[Size*Size];
    CopyData(pMatrix, Size, pSerialMatrix);
  }

  // Data distribution among the processes
  DataDistribution(pMatrix, pProcRows, Size,RowNum);

  // Paralle Gauss-Seidel method
  ParallelResultCalculation(pProcRows, Size,RowNum,Eps, Iterations);
  //TestDistribution(pMatrix, pProcRows, Size,RowNum);

  // Gathering the calculation results
  ResultCollection(pProcRows, pMatrix, Size, RowNum);
  TestDistribution(pMatrix, pProcRows, Size,RowNum);

  // Printing  the result
  printf("\n Iter %d \n", Iterations);
  printf("\nResult matrix: \n");
  if (ProcRank==0) {
    //TestResult(pMatrix,Size,pMatrixCopy,Eps);
```

```
    PrintMatrix(pMatrix,Size,Size);
  }

  // Process termination
  if (ProcRank == 0) delete []pSerialMatrix;
  ProcessTermination(pMatrix, pProcRows);
  MPI_Finalize();
}
```